Brewster –

Susan Leinberger
415-354-0263
called about setting
up appt. (Steve Jobs)
2:30pm Fri

Privacy in the age of ~~Com~~ Parralled computers

A computer revolution is taking place that will bring the cost
of large scale computing down 100-1000 times! due to the ~~[illegible]~~ use of parallel
processors, ~~Communications + lib~~. ~~These machines~~ Computers that use many cheaper processors "in concert" to achieve
tasks that single large processors could. The benifits of such a shakeup have
long been heralded.
can ~~compute~~ ~~achieve~~ afordable, run programs that could not be thought of before.

The effect on personal privacy has not been addressed in the
context of computers that can make dragnet searches through data
with the ease of single inquiries today. Our government ~~and~~
other institutions hold a tremendous amount of personal, incidental information
on us. only with these new machines can this information be ~~put to use~~ thought ~~to~~ be a threat to our privacy.
Information storage technology has already made a similar
revolution so the Data is "ready to go"

- Data already collected
- Survailance impossible to avoid
- Government will give away information
  therefore we have a threat from commercial sources
- Dragnet searches are survailance
- ~~Col~~

Information collection and keeping is the problem.
No matter what we say you can do with the data,
if the data is kept is will be used. (no more $ limitation)
(Information Act says what it will be used for, this is not good enough)
en

- incidental information
- arrest + not conviction records
- Data should have expiration dates
- search + seizure applies to data,
  so requesting data from other
  parts of Gov should require
  a warrent.

Digitized by the Internet Archive
in 2026 with funding from
Brewster Kahle

https://archive.org/details/partialpapers00unse

~~Effects of~~ parrallel processing + Personal Privacy

①

The recent revolution in computing speed will add another dimension to the problem of protecting personal privacy in a world of governmental and commercial databases. Parallel processors are reducing the cost of computing (~~sorting~~ ~~and~~ searching and processing of data) by factors of 100 and promise factors of 1000. Thus programs and policies impractical and unimaginable are becoming possible and inexpensive. Dragnet searches and widespread survailance are not only possible, but inevitable.

~~At~~ the previous revolution Storage technology has already made collection of huge databases of personal data possible. Legislation was introduced to handle this change, such as the privacy act. ~~and~~ ~~The~~ The high cost of processing, on the other hand, has kept these accumulated databases

difficult to use because of the cost of searching and accessing them. Parallel processing makes the cost of processing ~~a full dragnet search~~ every ~~each~~ personal record at once the same as accessing one record in the past. With this ability, database owners don't need "probable cause" to look into one's files; due to cost constraints, (s)he can peruse all files looking for suspect people.

Dragnet searches ~~and~~ through incidental information may be a boon for the law enforcement and credit industries, but poses a ~~[illegible]~~ significant change to our legal and commercial conduct.

~~Watching~~ Widespread Collection and searches of incidental information is surveillance which is only now ~~becoming~~ is ~~more~~ possible and inexpensive.

~~For aid in this problem.~~

To combat this problem, I suggest the following:

1. ~~Limit collection and storage of incidental information~~

2. Define incidental information as that which is not legally recognised (conviction records are not incidental, arrest records are; Addresses are not incidental, house appearances are; credit ratings are not incidental, Visa card records are)

1. Limit governmental ~~collection~~ and commercial collection and long term storage of incidental informations.

3. Add to the privacy statement on ~~all~~ government~~al~~ questionnaires when the data will ~~be~~ expire and be ~~expunged~~ deleted from the databases. In otherwords, state the lifespan of a piece of information at collection time.

We can not practically limit what processing people do with their data, but we can effectively limit what personal data can be collected and how long it can be kept.

Please consider the possibilities: if people are hassled by a particular police officer unduly, they can record such offenses, and others may be careful to avoid this officer.

other information such as home phone numbers, visa card #s, social se

~~Corporation~~ Socially conscious business ~~men~~ people might find this person unworthy of hire. Unfair you say? Are the ~~sh~~ existing systems any ~~less~~ more fair? Lets take the scenario one step further. On these black listed people might be such information as their social security numbers visa card numbers, home phone numbers, physical descriptions, family informa ~~and~~ ~~medical records~~ ~~bank account balances~~ etc. all of this is perfectly legal to colle This would give the inquisative person enough information to make the offender's life miserable. Of course, the offender would have the right to see his record and have the right to suggest changes.

IS this use wrong? Yes of course it is but this power rests in the hands of anyone with enough money to buy access. this ~~it~~ would give each person the same power that our government and large businesses have over us. ~~So~~ I have no reason to trust these institution any more than the private citizen. Such a Peoples Computer would give us, the people as much power as our institutions until they ~~start~~ ~~stop~~ see the evils in all of it and ~~started~~ ~~to~~ enact meaningful legislation to protect our privacy and integrity. Laws will not be made until those that make the law find it in their best interest. right now only the private man suffers.

tion

The common element ~~in all of this~~ is that these records on private citizens are being used by and for the government and industry. I suggest we make a database for the people to be used by the people. With such a system, people could collect evidence against arogant cops

Imagine a police officer pulling you over for speeding and his calmly informing you, before you say anything, that your political beliefs, credit record, and ~~d~~ service discharge will reflect badly on you in a court case over the charge! ~~about your speeding just now.~~ ~~Such information is being collected on you by~~

Does this sound impossible? Such treatment is ~~It happens~~ already possible ~~now~~ in the UK + West Germany and is being set up in this country.

Right now computer records on you are being kept all over the contry by our government and industry. These records include your credit history, where you work, what magazines you subscribe to, what kind of car you own, what religion and political party you belong to, and what magazines you subscribe to etc. Often all one needs is your Social Security Number to find out about you. The list of information goes on. Most people I talk to say proudly, "Thats okay, I have nothing to hide." But the fact is that information ~~is~~ gives people power, and power ~~can be~~ invites abuse.

Elements of a fault tollerant computer.

I will attempt to focus on the crucial issues in advanced processing architecture and derive some of the elements for a reasonable computer.

The primary issues in a processing systems are

Fault tollerance

The basics of my solution

modular: to allow failure of a small part to be isolated from the rest

Homogeneous for interchangeability

Fuses for suicide + killing

voting for global lines (clocks instructions)

be errors due to noise or malfunction.

Other problems require correct operation. Minsky and Paperts problem: is the number of objects odd is a good example. The counting must be done flawlessly.

The distributed work machines that I am talking about are not just perceptron machines but a broader class containing cellular automata, Hillis's Connection machine, Fahlman's boltzman machine, and Hopfield's memory. These all have the feature that any one piece failing will not require the machine to fail.

Hardware of a Fault Tollerant System.

Lets, for the moment, leave aside the notion of how a distributed work machine will be used specifically and concentrate on how to make one that is fault tollerant. The necessary features are:

store state
Send information to others
recieve information
recieve energy
dissapate power
(optional) recieve global information

Any of these features may be lacking in a ~ a faulty component. Please note that I am not trying to distill the essence of intelligence; this is only a system model.

In current distributed work machines, global information is recieved by the workers. This requirement does not seem necessary but should be included in our list of "protected features" so that this hardware can be used for these machines.

Fault Tollerant machine

Fault tollerance is the ability for a system to function with ~~with~~ intermittent, and permanent ~~errors~~ faults. ~~Faults~~ can and do occur in any number of part in ~~of~~ large systems, so systems ~~their~~ ability to work around such problems will appropriate its performance in the long run. ~~[illegible]~~ ~~[illegible]~~ an ~~an~~ example of a successfully fault tollerant design is nature's tree, but will persist with a limb or without water for a while. As computer systems grow, the necessity to design fault tollerant systems ~~[illegible]~~ becomes imperative. This paper will discuss fault tollerant systems in general and suggest an architecture that exploits some of these features.

Faults: ~~[illegible]~~ Faults, ~~[illegible]~~ are [illegible] occurances or attributes that ~~[illegible]~~ detracts from system integrity. This is broad but it needs to be to cover power glitches, broken wires, and erroneous data, which plague computers today leading to unreliable systems. Wires are prone to glitches from other wires, being broken, being almost broken, contacting another wire etc. Memory is prone to not rembering, sometimes rembering from any number of causes. ~~[illegible]~~ Faults are a part of any system, so we must understand them to deal with them.

~~to~~ Faults can be ~~[illegible]~~ lumped into two groups, intermittent and continuous. Intermittent are classically harder to find since their cause is unknown therefore it is not repeatable. These ~~errors~~ might be a nuclear event that causes a bit of memory to be changed, ~~an~~ a local power ~~[illegible]~~ surge that changed the voltage on a wire, or someone walking by and giggled a contact.

For a computer system, this can mean that a program will crash, a wrong answer will result, or a database will permanently be wrong.

Permanent errors ~~[illegible]~~ occure repeatedly and predictably if one can recreate the right circumstances. The hardest hardware bug a friend of mine found was when a power pin on a NAND chip was bent so that if no inputs were a 1, then the ~~system~~ chip didn't have the power to answer a 1. The gates were used in different parts of the CPU so the bug seemed intermittent and random. On a tree this might correspond to a missing branch, ~~[illegible]~~ leaves in the shade, or dry soil.

Dealing with errors:

Faults are nothing new, neither are solutions, so I will describe a few

Redundancy: Use several identical systems to serve as backups for each other. ~~etc~~ Humans have two eyes two hands etc. This requires a least a complete system for each expected fault.

Error correction: Include enough information or power to detect and interpolate ~~to~~ for an error. On one of the Viking landers a fourth gyro was included as a backup. Sure enough as it was landing, one primary one froze, so the fourth was used in conjunction with the other three to figure out the ~~fourth's~~ third's information. Error correcting codes are another example. This requires less replication for the same reliability as the redundancy system, though it requires a working controller to make decisions.

Distributed work: In such a system, if 10% of the system failed, it works 10% ~~less~~ more slowly or less powerfully, 25% would lead to 25% loss. A dam might work this way to generate power; if any turbines get stuck then they don't produce, but the others do. A tree's chlorophyll producers work that way, if a leaf drops off, than just that loss is felt. Catastrophic events happen when a large percentage of a system is bad. ~~at~~ What percentage this happens can be an estimate of its ~~st~~ robustness.

This last system of distributed work seems like the best system methodology, let us examine it further. Many large systems work this way: a city is a group of buildings any several of which could fall down without causing all of the buildings and other functions to stop working. Forests can have a disease of one type of tree and the rest will live.

Here are some possible generalizations components in ~~that~~ systems that exhibit distributed work:

- modular/granular
- partially independent ~~ex~~
- similar though not identical
- parallel goal which accept "contributions" from components.

Fault tollerant computers: Currently there are computer systems that ~~exist~~ a tollerant by redundancy and error correction. Tandem computer makes systems where a few ~~several~~ identical computers are sitting next to each other computing the same instructions. ~~and checking the~~ Each computer compares its answer to ~~each~~ the others ~~and~~ to decide the right answer. If one chip blows, or one wire comes loose then an entire mainframe will crash but the computation will continue to be correct. In such a system ~~the fact that~~ one error in one transistor can make millions useless. ~~[illegible]~~ This ~~pro~~ technic is expensive per fault. If there were three computers voting then three errors, badly placed, would bring down the whole system.

Error correction in computers is used ion communication lines. With a message can be sent a small add-on that could lead to the correction of minor errors in the message. This requires that large failures don't ~~exist~~ occure and that the correcting machinery is working.

As far as I know, no computer uses distributed work for preserving integrety though certain architectures are natural for it. Computers that are sometimes called "Massively Parallel" *could exploit many of the features of such fault tollerance since they exhibit some of the characteristics of natural Distributed work systems.

Computer Power to the People

Right now, computer records are being kept ~~all over the~~ country on ~~people~~ you including whether ~~who have~~ you've have bounced a check, been arrested, own a car, go to church, subscribed to ~~certain~~ magazines* ~~along with~~ ~~etc~~ ~~on co~~ computers all over the country. Databases are ~~the~~ big business in the government and business community. For example, The IRS is ~~are~~ trying to secure magazine subscription lists, local police have access to FBI records, and the selective service bought ~~[illegible]~~ state drivers licence records, etc. the coordination of information has become an obsession in government and industry. This information, ~~[illegible]~~ correct or not, is being used to decide your credit, jobs, and legal status.

* along with where you work, how much you make, ~~what~~ what political party you ~~vote~~ ~~reg~~ registered for.

The common element in all of this is that these records ~~are being~~ used on private people are being used only by large organizations I suggest we have a database for the people. ~~this database would be operated for private individuals~~ Such a service would provide an opportunity to the ~~[illegible]~~ common person to warn others about corrupt police, bad judges, nasty IRS officials etc. Anyone could add to it ~~act~~ by dailing up the computer or sending it in. Such data might include accounts of events, how to avoid the person etc. ~~home phone numbers, visa card numbers, home addresses~~. Since all are force to look at it over a telephone line, this can add everyone in their ~~go~~ quest for knowledge about their government employees.

This occurred to me after, essentially, being arrested for not giving my SS number
9/83

The People's Computer

We need an open computer system that anyone can access and add to. This would be available over phone lines and by mail.

Purpose: TO keep records on people that are enemies of the people. This includes police, IRS, judges etc who have done people wrong. This could include such information as phone numbers, ~~so~~ SS numbers home addresses, descriptions of events.

Justification: These records are used against us, lets use it on them. It is within the law, except these records are usually kept only on ~~a~~ little ~~to~~ people by and for the government and big corporations.

Real use: Annoy the obnoxious power people, shock the powers that be into realizing the power of the computer.

Required: IBM PC or bigger, 10 m bytes disk reasonable data base system for it.

Problems: this is only evil, Only a negative comment, and might be used by truely violent groups of a hit list.

I/O on wafer $\propto$
3

assume standard pads only on perimeter

~~M~~ perimeter $= \pi D =$ 20 inches $=$ 50 cm

pad $=$ 150 uM on a side

~~pads/perimeter~~:

pads/wafer $=$ ~~$\frac{150\times10^{-6}M}{50\times10^{-2}M}$~~ $= \frac{50\times10^{-2}M}{150\times10^{-6}M} = \frac{1}{3}\times10^{4}$

$=$ 3K pads/wafer

~~assume~~ How to bond these to pins

pins are .1 inches apart $=$

.25 cm apart

$\therefore$ $(.25\text{ cm})^2$ area

3K pins x Area/pin $=$ Area of all pins

$3\times10^{3} \times 6\times10^{-6} M^{2} = 18\times10^{-3} M^{2}$

Area of wafer $= \pi(.075 m)^2 = 17\times10^{-3} M^{2}$

Takes the whole wafer to put them into a PC board.

Not practical

~~3K pads/wat.~~

How ~~to~~ many dimensions is that?

figure 23K processors,

1K routers, takes 1K pins/dimension

$\therefore$ 3 dimensions. Not great

With Blinking

consider another wafer mounted next to it, Current Blinkers (Japan Electronics magazine Dec 15, 1983) are ~~20~~ 210 × 400 μM. Detectors are small assume 20 × 20 μM

say spacing is 5 times distance between wafers.
wafers can be .1cm apart ∴
.5cm between each ∴
Area ~~between~~ of blinkers is $.25 \times 10^{-4} M^2$

$$\text{Blinkers/wafer} = \frac{50 \times 10^{-2} m^2}{.25 \times 10^{-4}} = \frac{5 \times 10^{-1}}{2.5 \times 10^{-5}} = 2 \times 10^4$$

$$= 20\text{ K Blinkers}$$

$$\text{Blinkers/router} = \frac{32 \times \text{Area of Proc}}{\text{Area of Blinker}} = \frac{32 \times 375 \times 10^{-9} m^2}{2.5 \times 10^{-5}}$$

$$= .5 \text{ Blinkers/Router}$$

one more dimension ∴ can make twice as large a machine without decreasing speed.
∴ 43 K machine on 2 wafers

Distributed Work

What kind of problems can be solved by distributed work.

Search, can be done in parallel
building, such as an anthill by ants
if more than one piece can
be added at a time.
matching, if mostly local information
is used to make the decision

How can't anything be done if workers make sporatic and permanent mistakes? Success can not depend on one of the components. Sometimes this is easy like in perceptron like edge detection, where each "worker" looks at a set of retina points. These regions are overlapping to prevent an edge sneaking in between the workers. These workers detect a sharp change and send their response to an upper level worker, this may then ask neighbor workers for confirmation and forward that answer etc. If one worker is bad then it is unlikely to cause catastrophic errors. If an edge is found to extend for inches ~~[illegible]~~ except for a few points, then these points can

Parallel Code Generation for the Connection Machine II

Brewster Kahle
Margaret St. Pierre
Thinking Machines Corporation
Cambridge, Massachusetts

August 3, 1989

Abstract

A parallelizing compiler piped to a front-end serial compiler can produce efficient code for a data-parallel machine. This paper describes the parallel code generation phase of the parallelizing compiler for the Connection Machine Model II architecture. The novel splitting and merging techniques used in the parallel code generation phase are applied to the compilation of data-parallel programs. The source language is the data-parallel language called LoC.

1 Introduction

The Connection Machine 2 is a data parallel computer that is designed to run data intensive applications. A Connection Machine system consists of a conventional serial machine as a front end, many tens of thousands of data processors, and a selection of I/O devices and processors. A program executing on the front end commands all the components of the Connection Machine system including the data processors, disk system, and I/O devices. The user program is in many ways similar to a conventional serial machine program using UNIX like devices, but at run time many interconnected computers are used to perform operating system functions.

Programming a data parallel computer is more akin to programming serial machines than to programming control parallel machines. Data parallel programs have only one control sequence, and the program executes on one processor, the front end. Thus, the program is running in a familiar environment with familiar tools. The data resides in Connection Machine memory and is manipulated by the CM data processors, which can access the memory of other CM processors by using a high speed intercommunication network. Similarly the front end can access the CM memory easily and efficiently. A major difference between data parallel and serial code is that iteration over the data objects is unnecessary, as all data objects can be operated on at once.

A Sun Microsystems®Sun-4®workstation can be used as a front end to the Connection Machine processors in running user programs. This processor controls resource allocation, performs scalar computation, and executes the control portions of Connection Machine programs. No programs are stored or executed in the CM data processors. The user programs run on the front end and issue instructions to the Connection Machine. Data may also be transferred between the front end and the Connection Machine processors via this interface.

2 Runtime System

Field Manager, Geometry Manager, and VP Manager: Paris style field manager, geometry manager, and vp manager

Loader: manages the Control store of the sequencer (no scratch ram) Costs one memory reference if the u-function is loaded

User manager: handles the registering of multiple users records address space, timesharing data accounting data etc

3 Overview of the Compilation Process

The compilation of a LoC program occurs in two basic steps. In the first step, the parallelizing compiler takes in the data-parallel LoC source code, compiles only the parallel pieces, embeds the compiled pieces back into the serial code, and constructs a serial C target program. In the second step, a machine-specific front-end compiler takes the resulting serial C program, and compiles it into object code to be executed by the front-end machine. The front-end compiler is any standard C compiler that runs on the front-end machine.

```
                The Compilation Process

              +-----------------+
LoC     -->   |  Parallelizing  |
Program       |    Compiler     |
              +-----------------+
                      |  Serial
                      V  C program
              +---------------+
              |  C Compiler  | --> Object Code
              +---------------+
```

Since the front end of the parallelizing compiler is typical of standard serial compilers, the next section is devoted to the parallel code generation phase of the parallelizing compiler.

4 Parallel Code Generation

The *parallel code generation phase* of the parallelizing compiler takes in intermediate code and generates serial C code. As in most conventional compilers, the intermediate code is first broken up into a set of basic blocks. Unlike compilers for serial machines, in each basic block the parallel code is separated from the serial code. The parallel code then goes through register allocation, during which additional serial code may be created. Next, microcode generation and assembly occur on the parallel code. The resulting parallel object code and the serial code created during register allocation is merged back into original the serial code, and C is generated. More specifically, parallel code generation occurs in six[1] stages:

1. dividing the code into basic blocks,
2. splitting,
3. register allocation,
4. microcode generation,
5. merging, and
6. back-to-C code generation.

Each stage is discussed in detail in the following subsections.

```
          The Parallelizing Compiler

          +---------------+
LoC --->  |  front end    |
          +---------------+
                  |
                  V
          +-----------------+
          |    parallel     |
          | code generator  |
          +-----------------+
                  |
                  V
             Object Code
```

---

[1] Optimizations have been purposely omitted because they may occur during one or more of these stages.

A simple LoC function is shown below. The function performs a serial and parallel addition, compares the results, prints a warning message if the results do not match, and returns the parallel result. The variables **a**, **b**, **id**, **actual_result**, and **expected_result** are serial, and thus exist only on the front-end machine. The parallel variables **parallel_a**, **parallel_b**, and **parallel_result** exist in the processing elements, and have field addresses in the front-end machine. Since memory management for the processing elements is performed by the front-end machine, the front-end manages the field addresses of all parallel variables.

```
parallel_plus (a, b, id)
int a, b, id;
{
   int actual_result, expected_result;
   int :physical parallel_a, parallel_b;
   int :physical parallel_result;
   parallel_a = a;                                    (1)
   parallel_b = b;                                    (2)
   parallel_result = parallel_a + parallel_b;         (3)
   expected_result = a + b;                           (4)
   actual_result = [id]parallel_result;               (5)
   if (expected_result != actual_result) {            (6)
      fprintf(stdout, "Expected %d, got %d !\n",
              expected_result, actual_result);
   }
   return(parallel_result);                           (7)
}
```

Lines (1) and (2) broadcast a serial variable from the front-end to the processing elements; *i.e.*, a serial-to-parallel operation. Lines (3) and (4) perform a serial and parallel addition, respectively. Line (5) is a parallel-to-serial operation where the parallel variable **parallel_result**, contained in the processing element selected by the serial variable **id**, is assigned to the serial variable **actual_result**. Line (6) checks the serial against the parallel result, and line (7) returns the parallel result.

Once the LoC program passes through the front end phase of the parallelizing compiler, the intermediate code is passed to the parallel code generator. For simplicity, the intermediate code is represented as a linearized set of quadruples of the form,

op dest src1 src2

where *op* is the operator, *dest* is the destination field, and *src1* and *src2* are the source fields. To represent both parallel and serial intermediate code, the operators are identified by one of four classes:

- *Strictly Serial*: The source and destination fields are serial. The operation occurs entirely in the front-end machine.
- *Strictly Parallel*: The source and destination fields are parallel. The operation occurs entirely in the processing elements.
- *Parallel-to-Serial*: The source is parallel and the destination is serial. This corresponds to a reduction operation, *e.g.*, taking the maximum of a parallel variable, or reading a value from a processing element.
- *Serial-to-Parallel*: The source is serial and the destination is parallel. This corresponds to broadcasting a single value from the front-end machine down to all the processing elements.

The **parallel_plus** LoC function is shown below in linearized quadruple form. The operator class for each newly introduced quadruple operation is also noted.

```
(BEGIN-PDEFINE PARALLEL-PLUS INT          ; serial
     ((ARG A INT)
      (ARG B INT)
      (ARG ID INT)))
(TEMP PARALLEL-PLUS-RETURN-VALUE-1 INT)   ; serial
(TEMP PARALLEL-PLUS-TOP-OF-STACK-1 UINT)
(TEMP _T1 PINT)
(TEMP _T2 PINT)
(TEMP _T3 PINT)
(TEMP _T4 INT)
(TEMP _T5 INT)
(NEXT-STACK-FIELD                         ; serial
     PARALLEL-PLUS-TOP-OF-STACK-1)
(BEGIN-PLET)                              ; serial
```

```
(TEMP PLET-TOP-OF-STACK-1 UINT)
(TEMP ACTUAL-RESULT INT)
(TEMP EXPECTED-RESULT INT)
(TEMP PARALLEL-A PINT)
(TEMP PARALLEL-B PINT)
(TEMP PARALLEL-RESULT PINT)
(NEXT-STACK-FIELD PLET-TOP-OF-STACK-1)
(ALLOCATE-STACK-FIELD PARALLEL-A 4)     ; serial
(ALLOCATE-STACK-FIELD PARALLEL-B 4)
(ALLOCATE-STACK-FIELD PARALLEL-RESULT 4)
(PINT-P _T1 A)                ; serial-to-parallel
(PINT-PASSIGN PARALLEL-A _T1)           ; parallel
(PINT-P _T2 B)
(PINT-PASSIGN PARALLEL-B _T2)
(PINT-P+ _T3 PARALLEL-A PARALLEL-B)     ; parallel
(PINT-PASSIGN P-RESULT _T3)
(INT-+ _T4 A B)                         ; serial
(INT-ASSIGN EXPECTED-RESULT _T4)        ; serial
(INT-PREF _T5 P-RESULT ID)     ; parallel-to-serial
(INT-ASSIGN ACTUAL-RESULT _T5)
(BEGIN-IF IF-CONDITION-1)               ; serial
(BEGIN-PLET)
(TEMP PLET-TOP-OF-STACK-2 UINT)
(NEXT-STACK-FIELD PLET-TOP-OF-STACK-3981)
(PROCEDURE-CALL FPRINTF STDOUT          ; serial
 "Expected %d, got %d !\\n"
EXPECTED-RESULT ACTUAL-RESULT)
(DEALLOCATE-STACK-FIELD-UP-THROUGH      ; serial
 PLET-TOP-OF-STACK-2)
(END-PLET)                              ; serial
(END-IF)                                ; serial
(RETURN PARALLEL-PLUS-RETURN-VALUE-1)   ; serial
(DEALLOCATE-STACK-FIELD-UP-THROUGH
            PLET-TOP-OF-STACK-1)
(END-PLET)
(DEALLOCATE-STACK-FIELD-UP-THROUGH      ; serial
            PARALLEL-PLUS-TOP-OF-STACK-1)
(END-PDEFINE PARALLEL-PLUS)             ; serial
```

4.1 Basic Blocks

A function block is divided into a set of basic blocks separated by serial control flow. Serial control-flow operators, called *separators*, separate one block from the next. Each block contains at least one parallel or serial-to-parallel quadruple. If a separator quadruple is encountered prior to any parallel or serial-to-parallel quadruples, a new block is not created. In the **parallel_plus** example, only one basic block is formed.

The decision to divide blocks on control flow boundaries is due to the handshaking between the serial code and the parallel code. For example, during a serial-to-parallel operation, when a serial value is broadcast from the front-end to the processing elements, the front-end must send the value, and the processing elements must pick up the value. These functions are represented in the serial code as a **push**, and in the parallel code as a **pop**. The opposite is true when parallel values are read from the processing elements back up to the front-end; the serial and parallel code contain a **pop** and **push**, respectively.

In general, for each **push** in the serial code, there must be a corresponding **pop** in the parallel code, and vice versa. To gaurantee this, the blocks must be separated across serial control flow boundaries, An example of this behavior is described further in Section ??

4.2 Splitting

Splitting takes one basic block of quadruples and separates them into a serial sub-block containing serial and parallel-to-serial quadruples, and a parallel sub-block containing parallel and serial-to-parallel quadruples. The position of each of the parallel and serial-to-parallel quadruples within the original block must be remembered for subsequent merging. Below is the parallel sub-block derived from the **parallel_plus** example.

```
(PINT-P _T1 A)
(PINT-PASSIGN PARALLEL-A _T1)
(PINT-P _T2 B)
(PINT-PASSIGN PARALLEL-B _T2)
(PINT-P+ _T3 PARALLEL-A PARALLEL-B)
(PINT-PASSIGN P-RESULT _T3)
```

4.3 Register Allocation

Next, standard register allocation techniques are performed on the extracted parallel sub-block. Register allocation results in a set of serial quadruples and a set of parallel quadruples. For serial entities required by the processing elements, the front-end is required to broadcast, or

`push`, the entity down to the processing elements, and the processing elements pop the value. This handshaking requires a serial quadruple for pushing, and a parallel quadruple for popping.

If the lifetime of a parallel variable extends beyond the sub-block, *i.e.*, the variable is used as a source in a subsequent parallel sub-block, the field address of each parallel variable must be broadcast from the front-end machine down to the processing elements. Similarly, any serial variables present in a serial-to-parallel quadruple must also be broadcast. More specifically, to broadcast a value, a serial `push` must be generated in addition to a parallel `pop` and `load`. For example, the serial-to-parallel quadruple,

```
(PINT-P _T1 A)
```

may result in the serial quadruple,

```
(INT-PUSH A)
```

and the parallel quadruple,

```
(INT-LOAD RO (POP))
```

which essentially stores the value of A into register RO in all the processing elements. If it were necessary to load or store `T1` into processing-element memory, the field address of `T1` could have also been broadcast from the front-end, and a `pop` followed by a `store` operation could have been done in the processing elements. To guarantee correct behavior, each asynchronous `push` performed by the front-end machine must occur in the same order as each asynchronous `pop` performed by the sequencer.

When register allocation is complete, any parallel variables used as a source in subsequent quadruples outside the parallel sub-block must be stored out to memory. To eliminate memory references, across-block optimizations can be added since the order of execution is known by control flow analysis.

4.4 Microcode Generation

After register allocation and optimizations are performed, parallel microcode, called a *microfunction* is generated for the parallel sub-blocks. Assembly code can be generated in a single pass because no control flow exists in the parallel sub-blocks.

4.5 Merging

Merging is the process of inserting the microfunction definition, the microfunction call, and the serial pushes constructed during the register allocation phase, into the serial sub-block. The microfunction definition, which creates an array of microinstructions is placed after the function definition quadruple at the beginning of the block. The microfunction call is placed just before the position of the first parallel quadruple. This call invokes the loader at runtime; the loader, in turn, makes sure the microfunction has been loaded into sequencer memory. And finally, the serial pushes are inserted in the position of the original parallel quadruples. Below is the result of the merging process for the `parallel_plus` example.

```
(BEGIN-PDEFINE PARALLEL-PLUS INT
 ((ARG A INT) (ARG B INT) (ARG ID INT)))
(DEFINE-U-FUNCTION SEQ-BLOCK-0 (0 3456 7962 3289 ...))
(TEMP PARALLEL-PLUS-RETURN-VALUE-1 INT)
(TEMP PARALLEL-PLUS-TOP-OF-STACK-1 UINT)
(TEMP _T1 PINT)
(TEMP _T2 PINT)
(TEMP _T3 PINT)
(TEMP _T4 INT)
(TEMP _T5 INT)
(NEXT-STACK-FIELD PARALLEL-PLUS-TOP-OF-STACK-1)
(BEGIN-PLET)
(TEMP PLET-TOP-OF-STACK-1 UINT)
(TEMP EXPECTED-RESULT INT)
(TEMP ACTUAL-RESULT INT)
(TEMP PARALLEL-A PINT)
(TEMP PARALLEL-B PINT)
(TEMP PARALLEL-RESULT PINT)
(NEXT-STACK-FIELD PLET-TOP-OF-STACK-1)
(ALLOCATE-STACK-FIELD PARALLEL-A 4)
(ALLOCATE-STACK-FIELD PARALLEL-B 4)
(ALLOCATE-STACK-FIELD PARALLEL-RESULT 4)
```

```
(EXECUTE-U-FUNCTION SEQ-BLOCK-0)
(INT-PUSH A)
(INT-PUSH B)
(PUSH-FIELD PARALLEL-RESULT)
(INT-+ _T4 A B)
(SERIAL-ASSIGN EXPECTED-RESULT _T4)
(INT-PREF _T5 P-RESULT ID)
(INT-ASSIGN ACTUAL-RESULT _T5)
(BEGIN-IF IF-CONDITION-1)
(BEGIN-PLET)
(TEMP PLET-TOP-OF-STACK-2 UINT)
(NEXT-STACK-FIELD PLET-TOP-OF-STACK-3981)
(PROCEDURE-CALL FPRINTF STDOUT
        "Expected %d, got %d !\\n"
EXPECTED-RESULT ACTUAL-RESULT)
(DEALLOCATE-STACK-FIELD-UP-THROUGH
        PLET-TOP-OF-STACK-2)
(END-PLET)
(END-IF)
(RETURN PARALLEL-PLUS-ADD-RETURN-VALUE-1)
(DEALLOCATE-STACK-FIELD-UP-THROUGH
        PLET-TOP-OF-STACK-1)
(END-PLET)
(DEALLOCATE-STACK-FIELD-UP-THROUGH
        PARALLEL-PLUS-TOP-OF-STACK-1)
(END-PDEFINE PARALLEL-PLUS)
```

4.6 Back-to-C Code Generation

And finally, the resulting serial quadruples are translated back into C. At this point, the C code may be compiled using the front-end compiler. Because of the portability of the C language, the back-to-c code-generation strategy permits porting the parallelizing compiler to other front-end machines.

4.7 Virtual Processors

References

[DAP] Flanders, P.M. et al, "Efficient High Speed Computing with the Distributed Array Processor," *High Speed Computer and Algorithm Organization*, Academic Press, 1977, pp. 113-127.

[Seitz85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, January 1985.

[TMC86a] Thinking Machines Corporation, "Introduction to Data Level Parallelism," Thinking Machines Corporation Technical Report 86.14, April 1986.

[MPP] Batcher, Kenneth E. (1980). "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, C-29 (9).

[TMC88a] Douglas, Kahle, Vasilevsky, "The Architecture of the CM-2 Data Processor," Thinking Machines Corporation Technical Report HA88-01, April 1988.

[TMC87] Thinking Machines Corporation, "Model CM-2 Technical Summary", Thinking Machines Corporation Technical Report HA87-4, April 1987.

[BBN] Bolt Beranek and Newman Inc. "Development of a Butterfly Multiprocessor Test Bed," Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[OPT] Lane, T. A. et al, "Digital system applications of optical interconnections," *Proc. SPIE O-E/Fiber Lase*, September, 1988.

[NEC] Arnould, E. A. et al. "The Design of NECTAR: A Network Backplane for Heterogeneous Multicomputers," *Proc. Third Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, April, 1988.

UNIX and the Connection Machine Operating System

Brewster U. Kahle

William A. Nesheim

Marshall Isman

Thinking Machines Corporation
Cambridge, Massachusetts

ABSTRACT

The UNIX[1] operating system is used on several different computers that comprise the Connection Machine®[2] system. This makes the Connection Machine operating system a simple distributed operating system optimized for large data applications. Common minicomputers are used as front ends for the data processors, with one controlling the disk system, and other computers used as I/O processors. The front end acts as a center of control for the other parts of the system, and is connected to the data processors via an interface on its I/O bus. Programs run on the front end, and instructions are sent to the Connection Machine (CM) processors to read, write, and manipulate data in CM memory. CM file transfers are set up and monitored by the front end and the disk controlling computers while the data is transferred over a high speed I/O bus. Application specific data transfers are handled by other I/O processors over a general purpose interface to the high speed data I/O bus. UNIX is used as the operating system on these components to provide a unified operating system model for the system.

This paper will illustrate how the pieces of the Connection Machine System are integrated to provide a distributed operating system based on UNIX. The structure and the integration of the overall system will be discussed without going into detail on the design and implementation of each. Furthermore, a prototype method of timesharing a massively parallel machine will be outlined. Finally, a list of interesting issues still to be addressed in operating systems for massively parallel computers will be presented.

1. Introduction

The Connection Machine system is a data parallel computer that is designed to run data intensive applications. This system consists of a conventional serial machine as a front end, many tens of thousands of data processors, and a selection of I/O devices and processors. A program executing on the front end commands all the components of the Connection Machine system including the data processors, disk system, and I/O devices. The user program is in many ways similar to a conventional serial machine program using UNIX like devices, but at run time many interconnected computers are used to perform operating system functions. Controlling these computers requires a distributed

---

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] Connection Machine is a registered trademark of Thinking Machines Corporation.

operating system. This paper describes the Connection Machine operating system.

The Connection Machine operating system is a hybrid of UNIX and CM specific code that controls the components of the system. Since the CM is quite different from conventional computer systems, its computational model will be briefly described. An overview of the hardware and software components of the system are given, and the interface between UNIX and the various portions of the Connection Machine system are described in some detail. Further work on the Connection Machine operating system is suggested.

2. A Data Parallel Computer System

Two distinct types of parallelism can be found in today's parallel computers. *Control parallel* computers achieve increased performance by taking advantage of parallelism found in the control structure of programs. The Cray X-MP, BBN Butterfly,[3] and CalTech Cosmic Cube[4] are examples of control parallel computers. In these machines, each processor executes a portion of the program. Consequently, each processor must have capabilities comparable to the processor of a serial computer on which the same program could be run.

Data parallel computers achieve increased performance by taking advantage of parallelism found in the data of a problem.[5] The Connection Machine system, DAP,[6] and Massively Parallel Processor[7] are examples of data parallel computers. Data parallel computers consist of a single instruction engine, and thousands of data processors, each having local memory and connected to a communications network over which they may exchange information with other processors. There are two reasons the factors affecting the design of data processors in a data parallel computer are quite different from those affecting the design of the processors in a control parallel computer. First, the control aspects of a program on the data parallel computer may be executed by the instruction engine. This means the data processors are not required to handle instructions, and may instead be optimized for data manipulation. Second, data parallel problems have tens of thousands of data elements which may be operated on simultaneously with minimal interprocessor interaction.

Programming a data parallel computer is more akin to programming serial machines than to programming control parallel machines. Data parallel programs have only one control sequence, and the program executes on one processor, the front end. Thus, the program is running in a familiar environment with familiar tools. The data resides in Connection Machine memory and is manipulated by the CM data processors, which can access the memory of other CM processors by using a high speed intercommunication network. Similarly the front end can access the CM memory easily and efficiently. A major difference between data parallel and serial code is that iteration over the data objects is unnecessary, as all data objects can be operated on at once.

3. Connection Machine System Components

The Connection Machine system can consist of several front end processors, a dividable block of data processors, DataVault[8] disk units, high speed graphic display systems, and various I/O computers. While many combinations are possible, this section will describe one configuration in order to illustrate the function of each component in the system (see figure 1).

---

[3] Bolt Beranek and Newman Inc. *Development of a Butterfly Multiprocessor Test Bed*, Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[4] C. L. Seitz, *The Cosmic Cube*, Communications of the ACM, Vol. 28, No. 1, January 1985

[5] Thinking Machines Corporation, *Introduction to Data Level Parallelism*, Thinking Machines Corporation Technical Report 86.14, April 1986

[6] Flanders, P.M. et al, *Efficient High Speed Computing with the Distributed Array Processor*, High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 113-127.

[7] Batcher, Kenneth E. (1980). *Design of a Massively Parallel Processor*, IEEE Transactions on Computers, C-29 (9).

[8] DataVault is a tradmark of Thinking Machines Corporation.

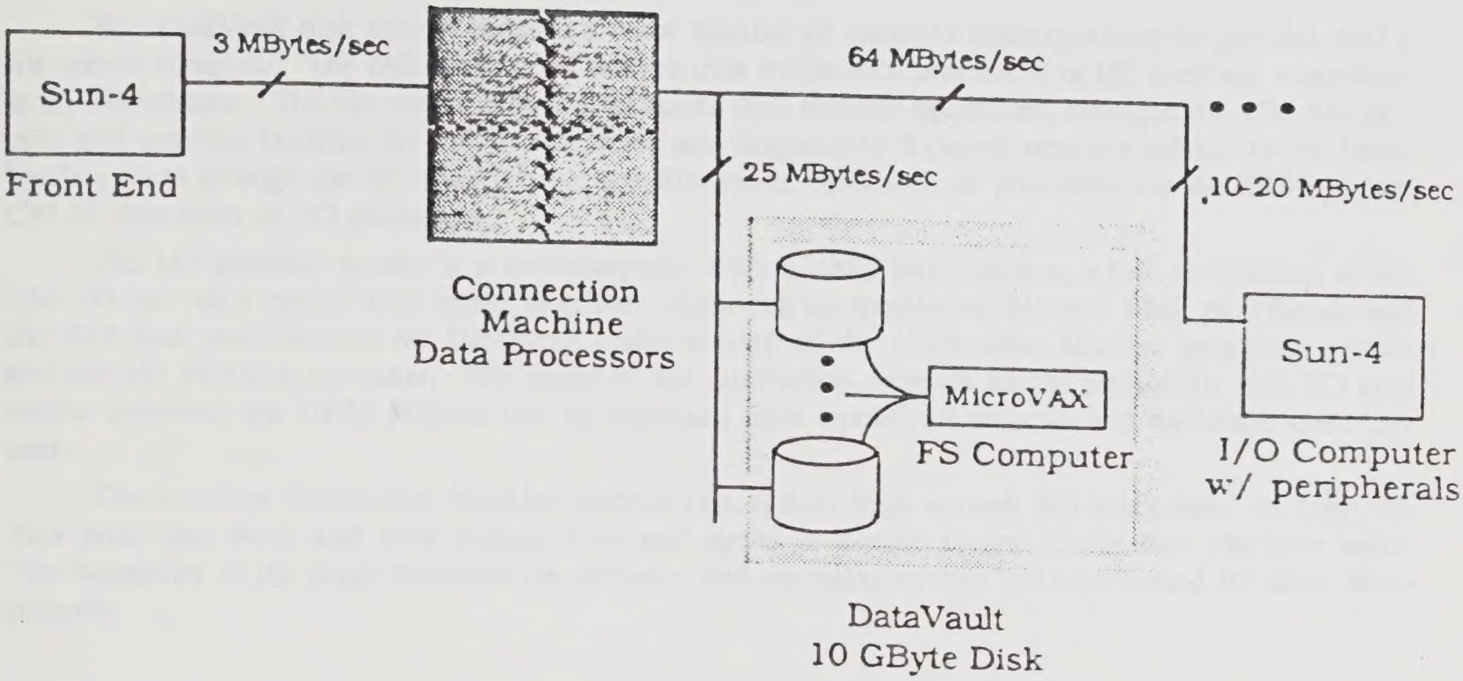


Figure 1: Example Connection Machine System

A Sun Microsystems®[9] Sun-4[10] workstation can be used as a front end to the Connection Machine processors in running user programs. This processor controls resource allocation, performs scalar computation and executes the control portions of Connection Machine programs. No programs are stored or executed in the CM data processors. The user programs run on the front end and issue instructions to the Connection Machine. Data may also be transferred between the front end and the Connection Machine processors via this interface.

The data processors store and process the larger data segments of an application. Each data processor operates on a different piece of data using the same instruction from the controlling front end processor. The data processors are very simple; a typical configuration consists of 16k - 64k processors containing 128 to 512 megabytes of memory.[11] A 64k processor machine performs a 4k by 4k matrix multiply at about 2500MFlops.[12] Each processor can efficiently access the memory of other processors and process the data within its local memory.

The connection between the scalar front ends and the data processors is a front end bus interface *(FEBI)*, a crossbar switch called the *Nexus*, and a *Sequencer* that takes macro-instructions and issues lower level instructions (nano-instructions) to the individual data processors. The FEBI is connected to the system I/O bus of the front end computer. Scalar transfers from the front end to the Sequencer run at approximately 2.5-3.5 MBytes/sec. Direct memory access block transfers are not supported on this interface because the bulk of the transfers are small instruction transfers from the front end CPU. The

---

[9] Sun Microsystems is a registered trademark of Sun Microsystems, Inc.

[10] Sun-4 is a trademark of Sun Microsystems, Inc.

[11] Thinking Machines Corporation, *The Architecture of the CM-2 Data Processor*, Thinking Machines Corporation Technical Report HA88-01, April 1988

[12] Thinking Machines Corporation, *Model CM-2 Technical Summary*, Thinking Machines Corporation Technical Report HA87-4, April 1987

Nexus connects up to 4 front ends to 4 sections of a CM to allow for flexible configuration. In our example system, we can assume that there is simply a direct connection from the Sun-4 to the CM. Data from the front end is written to a fifo in the Sequencer which, in turn, interprets these instructions calls or handles incoming data as appropriate. Results are relayed back to the front end through an output fifo. Each of these components also has a few status and configuration registers that are accessible from the front end.

The DataVault disk system consists a large number of standard disks operating in parallel, and a file server computer. The disk system can deliver data to the data processors or I/O interface computers at 25 MBytes/sec. The file server computer initiates data transfer operations, manages the CM file system, and provides facilities for disk maintenance and diagnostics. External data is available to the DataVault system through the I/O interfaces of the file server computer, or preferably via the 64Mbyte/sec CM I/O bus from an I/O processor.

The I/O interface system is a minicomputer with a VME bus interface, which is attached to the CM I/O bus via a special high speed interface. Data can be transferred between VME peripherals and the CM data processors or the DataVault under control of the Connection Machine operating system and the I/O interface computer. The speed of the connection depends on the peripherals and I/O processor involved, but 10-20 MBytes can be expected from optimized software and hardware combinations.

The resulting Connection Machine system can sustain high enough I/O bandwidths to keep the data processors busy, and have enough front end speed to handle several Connection Machine users. The remainder of the paper discusses the software and operating system tasks performed by these components.

4. The Connection Machine Operating System (CMOS)

The Connection Machine system operating provides many of the features found in a conventional operating system but implements them in a distributed manner. Parts of the CMOS run on each front end computer, the Sequencer, the file server computer and any other I/O interface computers. This section explains where the functions of resource allocation and management are performed. Further details on the components are discussed in later sections.

The CMOS functions generally are layered on top of the local operating system running on each computer within the CM system. Due to the number of different computers involved and the standard nature of the local operating system functions required, it does not make sense for the CMOS to be the native operating system on each computer in the CM system. Instead a common operating system, which is some variant of UNIX, is used on each computer. The front end computer is a Sun-4 running SunOS[13] or a VAX®[14] running ULTRIX[15], the file server computer is a MicroVAX running ULTRIX, and the I/O interface computer can be any VME based UNIX system. The Symbolics[16] Lisp Machine, as a front end, is the only non-UNIX computer that currently can be a part of the Connection Machine system. Each local UNIX is used to provide interprocess communication primitives needed to communicate among the distributed components of the CMOS. The local OS is also used to provide protection, local file storage, and switching between multiple local processes. The only modification to the local UNIX OS is a device driver for the CM specific device on each computer.

The *cmattach* command is used to obtain access to the CM system. This command is analogous to logging in to a UNIX system; once this operation is complete, use of the CM can begin. A *cmfinger* command, analogous to the UNIX command *who*, uses the same UNIX IPC mechanisms to list all current CM users.

---

[13] SunOS is a trademark of Sun Microsystems, Inc.

[14] VAX is a registered trademark of Digital Equipment Corporation.

[15] ULTRIX is a trademark of Digital Equipment Corporation.

[16] Symbolics is a trademark of Symbolics, Inc.

Process management and memory management are handled by a combination of the front-end computer and the Sequencer. All policy level decisions are handled on the front ends and some mechanism level functions are implemented by the Sequencer.

The Connection Machine File System (CMFS) provides a UNIX-like hierarchical file system with a programming interface that parallels the UNIX file system call interface. The file server runs on a MicroVax located inside the DataVault parallel disk subsystem. The file server daemon is an ULTRIX user process that takes advantage of UNIX facilities for path name resolution and communication operations to service CMFS operations. Portions of the file system code run on the front end computer, portions on the CM Sequencer, and portions on the file server computer. The front end and file server computers use UNIX IPC for communicating control information, and the high speed parallel CM I/O data bus is used for data transfers.

An I/O interface computer can also participate as one end of a file system transaction. This computer can act as a client of the DataVault, for transferring data to or from the disk subsystem, or as a server for the data needs of the CM data processors. This computer is interfaced to the high speed CM I/O bus via a VME interface board. This board allows a standard VME based UNIX host and all its peripherals to be a part of a CM system. The I/O interface computer uses UNIX IPC and other UNIX devices along with the CM specific hardware and software.

The CM graphics display consists of a CM specific board which resides in the CM backplane and a standard high resolution color monitor. The software necessary to display images runs on the front end and Sequencer. Access to this device is attained via a mechanism similar to that of *cmattach*.

Error handling facilities is distributed among all of the components of the CM system. There is communication among the components to pass error information and use of the underlying UNIX system to log errors and provide core dumps and debugging tools for user programs.

5. The Connection Machine Front End Subsystem

The control of the Connection Machine system is based in the front end processor. This section will describe the mechanism by which the front end system controls the CM data processors (see figure 2).

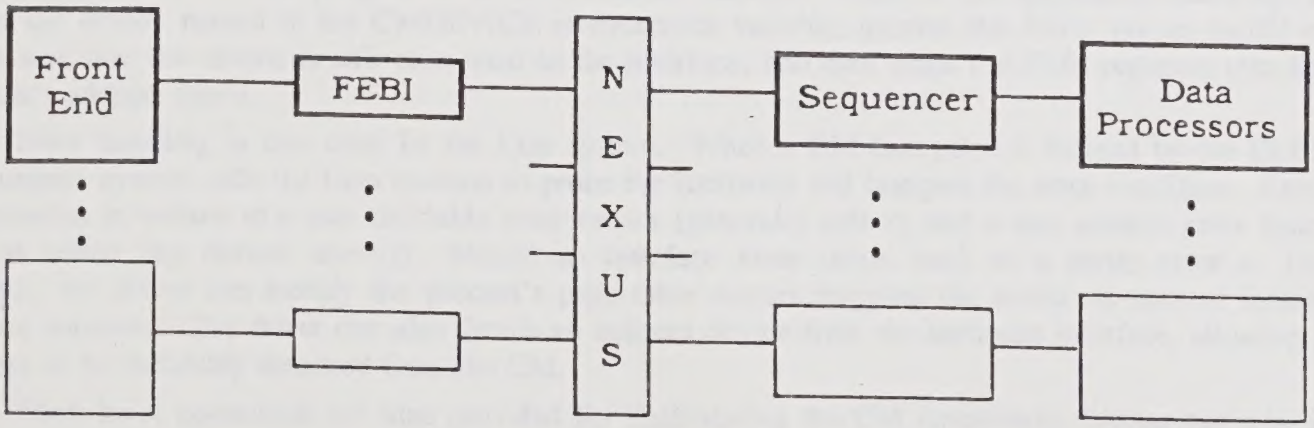


Figure 2: Front End Subsystem

The front end processor controls the front end bus interface (FEBI), the Nexus, and the Sequencer via 16 Connection Machine system registers. Four of these registers control the operation of the Nexus, while the remaining 12 access registers on the allocated Sequencer(s). Two of these registers are used to write to and read from fifo's in the Sequencer.

The ability to rapidly deliver instructions and get results from the Connection Machine is crucial to good application performance. For this reason the Sequencer registers are mapped directly into the address space of the front end system process currently "attached" to the Connection Machine system rather than using operating system calls to access the registers.

A process accesses CM resources by first gaining exclusive access to the FEBI. Access to the FEBI is controlled by the UNIX FEBI device driver. When a process desires access to the FEBI, it first opens an "indirect" CM device. This dummy device does not initially have access to the FEBI registers, and is in fact not associated with any hardware at all, but allows the user process to call driver *ioctl()* routines.

After successfully opening the indirect device, a process can issue a CONNECT_TO_INTERFACE *ioctl()* to request the driver to allocate a real FEBI. Blocking and non-blocking versions of this call take a list of requested FEBIs as an argument. Once successfully attached to a particular hardware interface, a process can map the FEBI CM registers into its address space. This mapping is accomplished on SunOS systems via a *mmap()* entry to the FEBI device driver, and via a special *ioctl()* in the driver on ULTRIX systems.

Gaining access to the Connection Machine processors requires the further step of setting up the Nexus to connect a given front end to the desired set of Sequencers and then initializing the CM processors, memory, and Sequencers. In some circumstances, new CM microcode is loaded into the Sequencers. For historical reasons, and for compatibility across our front-ends, the code which implements these functions is currently written in Lisp.

Users running Lisp under UNIX call C stubs to obtain access to the FEBI as described above. Then the Lisp process performs the Sequencer allocation and initialization functions. For non-Lisp access to the CM we have implemented a system which actually uses Lisp to do the allocation and initialization of CM hardware. The shell command *cmattach* calls the driver to access a FEBI. If one is not available, *cmattach* can optionally wait until a FEBI is available. Once the interface is allocated, a daemon process running a Lisp subprocess is called to set up and initialize the Connection Machine system. If the daemon indicates that the requested Sequencers are not available (for example if they are in use by a different front end), the cmattach command can sleep a short time and try the operation again.

Once the system is initialized, the cmattach program forks either an interactive sub-shell from which users execute their CM programs, or executes the user's program directly. The indirect device name is passed to the user program in an environment variable. When a CM application starts up, it opens the device named in the CMDEVICE environment variable, queries the driver via an *ioctl()* to make sure that the device is still connected to the interface, and then maps the FEBI registers into the process's address space.

Error handling is also done by the Lisp system. When a CM Exception is flagged by the FEBI, the runtime system calls the Lisp daemon to probe the hardware and interpret the error condition. Error information is written to a user definable error stream (generally *stderr*), and a user settable error function is called (by default *abort()*). Should an interface error occur, such as a parity error or bus timeout, the driver can modify the process's page table entries mapping the device to prevent further register accesses. The driver can also detach an indirect device from the hardware interface, allowing a process to be forceably detached from the CM.

Shell level commands are also provided for deallocating the CM *(cmdetach)*, finding out who is currently attached to or waiting for an interface *(cmusers)*, detaching another user or front end system from the CM *(cmdetach)*, and initializing the system *(cmcoldboot)*.

The front end system also acts as a center of control of CM I/O subsystem devices (figure 3). The simplest case is the Connection Machine graphics display system. The graphics display is connected to a interface board which resides in the Connection Machine system backplane. Under control of front end instructions, data is transferred directly from the Connection Machine processors to the display system. In order for a program to access the display system, the program must be attached to the section of the CM in which the frame buffer board resides. Thus allocation of graphics display systems is currently handled in the same manner as allocation of Sequencers and processors; the user either

gets exclusive access or no access at all.

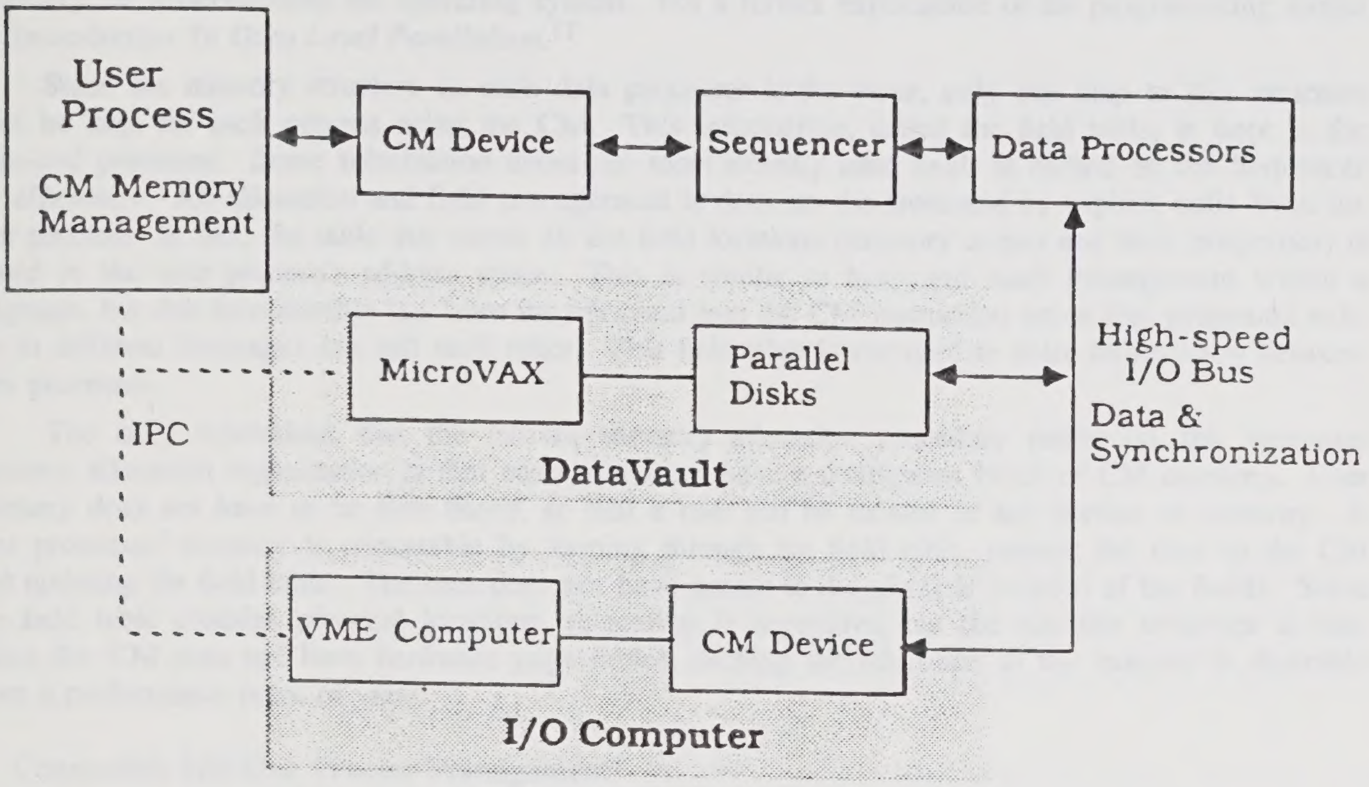


Figure 3: The front end and the I/O system.

I/O operations to the CM I/O bus are also initiated under front end control. Under front end control, the CM processors transfers data to the CM I/O controller board, and the I/O controller board is programmed to send or receive data on the CM I/O bus. Before a transfer on the bus is initiated, however, the front end system must arrange the other end of the transfer with the destination I/O subsystem controller host. These systems are are separate minicomputers running UNIX and in operations which transfer data between the CM and an I/O device, they operate in cooperation with the front end system.

When a process running on the CM needs to access data on the DataVault or retrieve data from a I/O processor, the front end system sends a command message to the I/O processor computer which then sets up the data transfer over the CM I/O bus. These command messages may be sent either via standard UNIX IPC mechanisms (TCP/IP networking), or via a command channel on the CM I/O bus. The I/O processor receives the message and passes it off to a CM I/O daemon server process. Another IPC message is sent back to the front end system to inform it of the success or failure of the operation. Data transfer is then initiated from the I/O processor to the CM via the CM I/O bus, freeing the I/O processor and the front end processor for other computations. Completion or error information is passed from the I/O processor.

6. User Process Memory Management

Memory allocation within a user process has some novel aspects that are appropriate for massively parallel machines. Data structures (memory fields) are allocated across all processors. This striping means that large arrays have elements across the entire machine. In fact, each element of a large array appears to have its own processor. Further, the dimensions of the data structures are not limited to the number of data processors because of a mechanism for simulating more processors. Currently, the CM does not support disk based virtual memory, so each user's entire memory space

must fit within the physical memory of the CM (.5 GBytes for a 64k processor machine). Memory allocation within a processor is performed by the run-time system in user specified increments rather than fixed pages. These increments are often small such as a 1 bit flag or a 32 bit number in each of the data processors. This section will address only those parts of memory allocation system that are necessary for understanding the operating system. For a further explanation of the programming model see *Introduction To Data Level Parallelism.*[17]

Since the memory structure in each data processor is the same, only one map to this structure must be kept for each process using the CM. This information, called the field table, is kept in the front-end processor. Some information about the most recently used fields is cached on the Sequencer for efficiency. All allocation and field management is done on the front-end by explicit calls from the user process. In fact, the table that stores all the field locations (memory stripes and their properties) is stored in the user process's address space. This is similar to heap and stack management within a language, but this functionality has been incorporated into the CM instruction set so that programs written in different languages can call each other. This field table is not used to share information between user processes.

The only restrictions that the current memory allocation procedure makes on the inter-user memory allocation organization is that each user must have a contiguous block of CM memory. User memory does not have to be zero based, so that a user can be located in any portion of memory. A user processes' memory is relocatable by looping through the field table, moving the data in the CM and updating the field table. The user does not have access to the physical location of the fields. Since the field table contains physical locations, relocation is expensive, but the run-time behavior is fast. Since the CM does not have hardware page tables, caching the addresses in this manner is desirable from a performance point of view.

7. Connection Machine Process Management

Currently, multiple users can access the Connection Machine system through a simple batch system. Under batch operation, a simple queue is maintained on the front-end and the data processors are completely reset between users. Each user gets sole use of the data processors, and all their associated memory and I/O devices. Under batch, only one front end process has access to the CM at a given time. Timesharing the Connection Machine between several users on the front end system can be achieved by allocating the FEBI device for short intervals to different processes as they are requested. Each process is given a fraction of the total CM memory, but when active, uses all the CM data processors. This section will describe this simple timesharing model.

Under a prototype timesharing implementation, each user is allocated a virtual CM with no interprocess communication, which is identical to the batch model of the machine with the single exception of having less memory available on each data processor (see figure 4). Timesharing of the data processors is done on top of the UNIX timesharing system of the front end. Only operations requiring the use of the CM data processors cause the system to determine whether the user can use the data processors at a given time. Manipulation of serial data, front end I/O, and user interaction all leave the CM data processors idle. Since most data parallel programs are actually a mix of serial and parallel code, most operations requiring the use of the CM data processors come in bursts. Because of this the CM process switching quantum can be quite large by traditional timesharing standards such as 1 second.

---

[17] Thinking Machines Corporation, *Introduction to Data Level Parallelism.* Thinking Machines Corporation Technical Report 86.14, April 1986

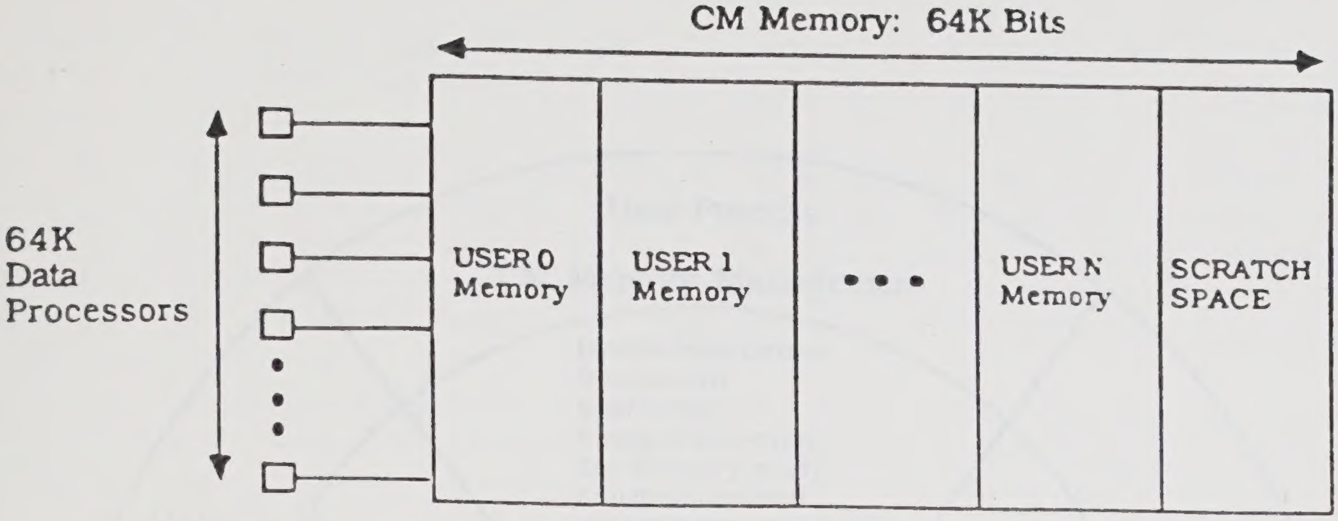


Figure 4: CM memory configuration under timesharing.

A prototype implementation of the CM timesharing system involves several pieces. Scheduling and resource management is handled by a daemon process. User processes requesting timesharing services use UNIX IPC primitives to communicate with the timesharing daemon. A structure is shared between the daemon and the user processes via UNIX shared memory facilities in order to implement an interlocking mechanism (see figure 5).

The timesharing daemon handles the registering of new users and switching between CM processes. A process gains access to the CM data processors by registering itself with the timesharing daemon. A UNIX IPC message is sent from the user process to the timesharing daemon requesting CM resources. The timesharing daemon sends a reply back to the user process indicating the success or failure of the request, and if successful, includes information on the amount of memory allocated to the user and a key for accessing a shared data structure. This data structure is used for most communication between the daemon and the user process.

The timesharing daemon can take the CM away from a user process anytime that process is not in a CM instruction. To implement this, all CM instructions check whether it is clear to run. If it is not clear to run then the process puts itself to sleep and waits for a signal from the daemon. If it is clear to run, then a flag is set in the shared data structure indicating that the user is actively using the CM. Once the CM instruction is completed, the flag is cleared. CM instructions are not interruptible by the daemon. To avoid a race condition, in fact, the instruction registers itself as using the CM before it checks to see if it has the right to actually use it, clearing the flag again if it is not clear to run. If the user process encounters an error on the CM it also gives up the machine, informing the daemon via another flag.

A further complication arises in handling errors that a user may have caused but not yet handled. A process may send many instructions to the CM before reading any results back. Error checking is generally done only at the time results are read back from the CM, so an error condition may exist for some time before a process notices it. In fact, one process may generate an error condition, and the CM switched to another process, before the first process is aware of the error. For this reason the timesharing daemon must check for any error conditions before switching CM processes. Since there are sophisticated error handling capabilities available to the user, an error does not lead to loss of state in the CM. The daemon uses another flag in the shared structure to indicate that an error has occurred in the background.

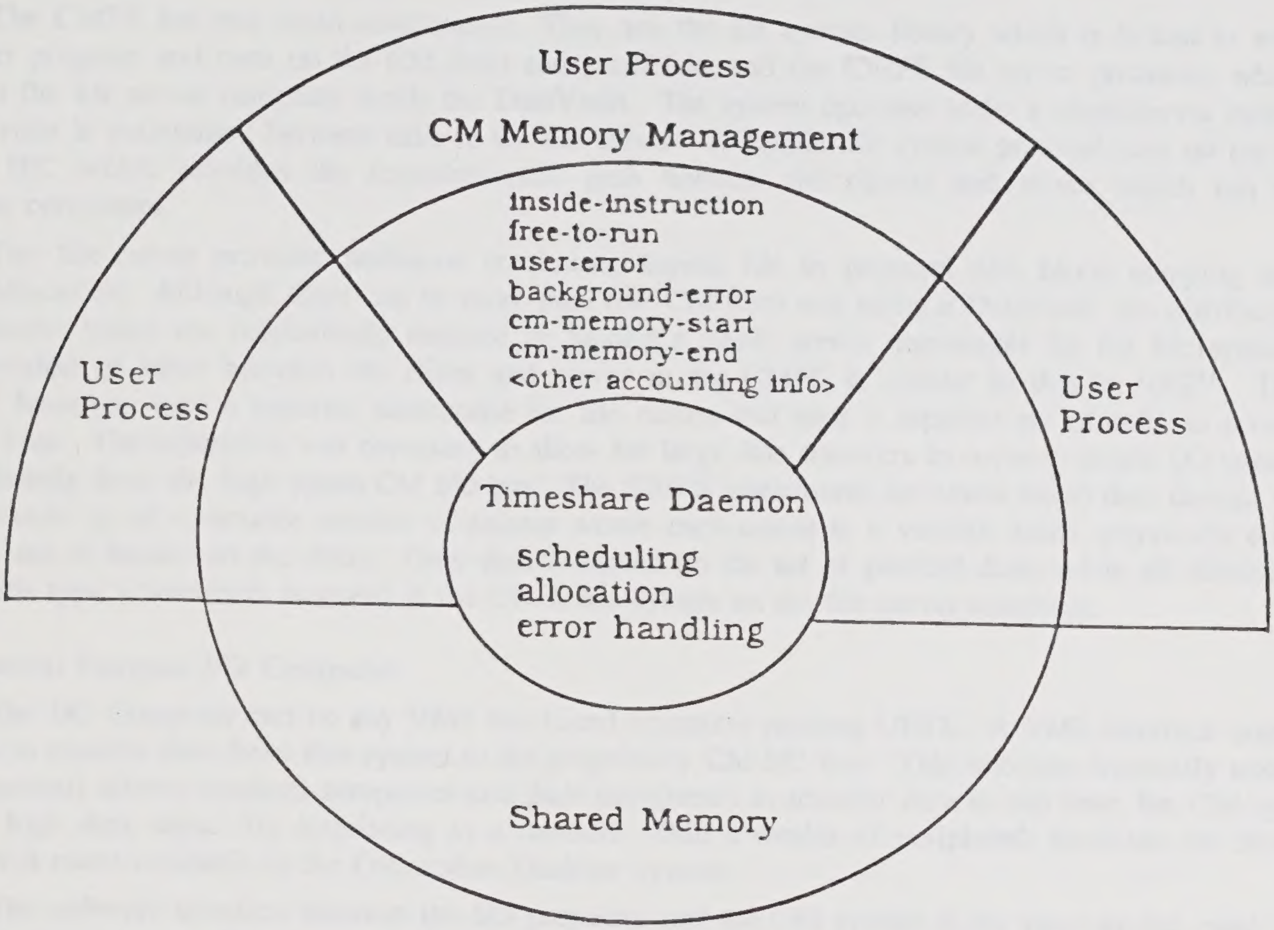

User Process
CM Memory Management
inside-instruction
free-to-run
user-error
background-error
cm-memory-start
cm-memory-end
<other accounting info>
User Process
User Process
Timeshare Daemon
scheduling
allocation
error handling
Shared Memory

Figure 5: Timesharing control mechanism.

When more than one process is registered with the timesharing daemon, the daemon begins operation as a scheduler. Access to the CM data processors is granted in a simple round robin mechanism, with the daemon granting a process access to the CM by setting the appropriate flag bit in the shared data structure, and sending the user process a signal if it is waiting for CM resources.

Inter-user protection is accomplished by a variety of mechanisms. User state in the sequencer is unloaded and loaded again by the timesharing daemon process. User state in the CM memory is protected by hardware bounds registers that keep users from altering each others memory (either accidently or otherwise). I/O operations are left to complete before processes are switched, thus achieving safety and simplicity while sacrificing some performance.

As in all operating system code, code ordering is important to prevent race conditions. Since the master is not in the kernel at this point, many of these considerations are exacerbated. CM instructions are simple writes to the FEBI registers, so it is extremely unlikely that the UNIX system will block the current CM process for an extended period of time. Timesharing on a massively parallel machine needs further design and implementation work.

8. The Connection Machine File System (CMFS)

The Connection Machine File System is a high speed, parallel, hierarchical file system. It runs on a parallel disk subsystem called the DataVault which stripes bits across 32 data carrying drives and places error correcting code information on 7 additional drives. The software and hardware is designed

to maximize throughput for large data transfers. The programming interface is very similar to the UNIX file system with the read and write primitives causing data to be transferred from each virtual processor in the CM.

The CMFS has two main components. They are the file system library which is linked in with the user program and runs on the CM front end processor, and the CMFS file server processor which runs on the file server computer inside the DataVault. The system operates under a client/server model where state is maintained between calls to the file server. A simple file system protocol runs on top of UNIX IPC which provides the communication path between the clients and server which run on separate computers.

The file server provides pathname resolution, logical file to physical disk block mapping and block allocation. Although there can be more than one CM front end using a DataVault, the distributed file systems issues are substantially reduced by having a single server responsible for the file system. The division of labor between the client and server in the CMFS is similar to that in NFS[18]. The CMFS, however, uses a separate namespace for file names and uses a separate set of calls to access CMFS files. The separation was necessary to allow for large data transfers to occur in single I/O operations directly from the high speed CM I/O bus. The CMFS implements an extent based data layout. A file is made up of a variable number of extents where each extent is a variable sized, physically contiguous set of blocks on the disks. Only data is stored on the set of parallel disks while all directory and inode type information is stored in the UNIX file system on the file server computer.

9. General Purpose I/O Computer

The I/O Computer can be any VME bus based computer running UNIX. A VME interface board is used to transfer data from this system to the proprietary CM I/O bus. This interface (currently under development) allows standard computers and their peripherals to transfer data to and from the CM system at high data rates. By interfacing to a standard VME a wealth of peripherals available for those systems is made available to the Connection Machine system.

The software interface between the I/O processor and the CM system is the same as that used by the CM File System. When moving data to and from the DataVault the I/O interface computer acts like the client and executes almost the same code as the CM front end computer executes when using the DataVault. The I/O Interface Computer can also act as the server when exchanging data with the CM. In both cases UNIX IPC and other UNIX devices are used for the operations. Since the I/O bus speed of the CM is 64MBytes/sec, this interface is limited only by the speed of the VME bus and the peripherals involved in the transfer.

10. Future Work on Massively Parallel Operating Systems

Operating systems for massively parallel computers, like the Connection Machine, open many design issues. Many of the performance characteristics for paging and user loads, for instance, are different for these systems. Thus the opportunities for significant performance and functionality increases are unfolding. Some of the areas that need further study are:

[1] Sophisticated Timesharing

[2] Virtual memory: Paging and Swapping

[3] Integrated I/O subsystems

[4] Remote Access

Timesharing systems could, for instance, be extended to have a process model that supported IPC, forking, and shared memory, but the specifics on how this would perform or even be designed for such systems is not understood. Further, asynchronous I/O might boost system throughput of a timeshared system. Another issue is the desirability of tightly coupling the timesharing system with the front-end kernel to make smaller Connection Machine time quantums efficient. An efficiently timeshared parallel computer would greatly extend the number of potential users and make very large systems more

---

[18] NFS is a trademark of Sun Microsystems, Inc.

affordable. Accounting and performance analysis on a machine like the Connection Machine is an interesting problem because of the number of independent processors working on one program.

Paging and swapping could extend the data sizes users can easily compute on from .5 GigaBytes to 10-100 GigaBytes. How the page tables should be managed needs further study of existing programs based on how memory access patterns of data parallel algorithms. Caching performance is also effected in interesting ways in the common data parallel applications.

Various input/output devices can be used to make data-parallel machines more useful. One can imagine the uses for high speed general purpose networks, frame grabbers, music synthesizers, speaker systems, FAX machines, and the like, if they were cleanly interfaced and controlled by a large compute engine. Keeping integrated control over a large number of different high speed devices, some of which might be physically distant, is an interesting task of distributed control.

Using a high speed computers as a compute server in workstation environments opens interesting issues of data transfer, simulation, and splitting of user programs. Many institutions will have large parallel computers accessible on networks of various performances (from 100MBytes to 56k Bits). How users can best use these resources is not understood. Handling this smoothly will put demands on networks and operating system code.

Many of the system performance characteristics are different enough in a massively parallel computer, that the many operating system issues need to be re-examined. We look forward to research in this area of operating system design to increase the usefulness and performance of massively parallel computer systems.

11. Conclusion

UNIX is an important vehicle for unifying the distributed parts of the Connection Machine system. Instead of implementing a native UNIX for the CM, we have used serial machine's implementations on different platforms and have built what we need on top of it. Massively parallel computers share many of the system demands that a serial machines do, so we can use the interfaces and philosophy of UNIX in implementing operating system components for the Connection Machine system.

Further work needs to be done to make the Connection Machine operating system a true multiuser, virtual memory system. Since the control issues and memory use statistics are different from serial machines, some of the current OS work can be exploited directly while other aspects need to be examined anew.

The Connection Machine Model CM-1 Architecture

Brewster A. Kahle and W. Daniel Hillis

Thinking Machines Corporation

Abstract

Massively parallel computer systems offer substantially improved performance over conventional supercomputers for many applications. Designed with many thousands of processing elements operating in parallel, these machines offer a high level of efficiency for computation-intensive algorithms. Machine architectures differ with regard to storage capacities and bandwidths, and these elements, along with software overhead, determine performance. This paper presents the Connection Machine®[1] Model CM-1, a massively parallel computer which exploits data parallelism for its performance. The performance of the CM-1 system is examined with the use of a simple application. Special consideration is given to the effect of problem size and data word size on performance.

1 Introduction

The purpose of this paper is to examine the performance of the Connection Machine Model CM-1 system, a general purpose computer capable of adapting to many sizes and types of data sets. It will provide an overview of the hardware and software systems, and present a simple application from which the raw performance of the machine can be examined. This

---

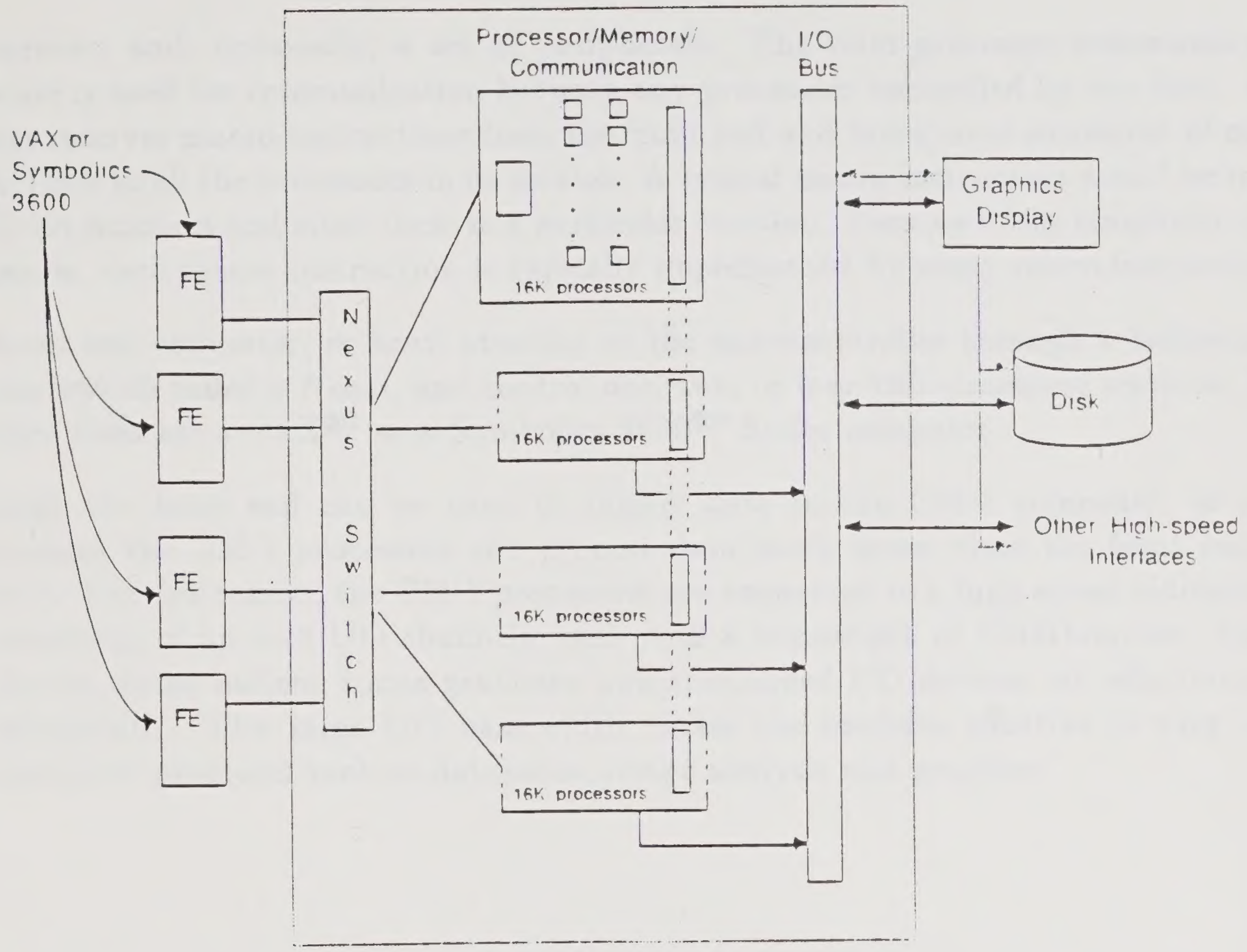
[1] Connection Machine is a registered trademark of Thinking Machines Corporation

will illustrate the power of data level parallelism and indicate how CM-1 performance is enhanced by the use of virtual processors and variable word size.

The Connection Machine system achieves its speed by operating on many thousands of data objects in parallel, rather than looping over them serially. The Connection Machine processors work with a conventional serial front-end computer as a controller, which issues the instructions executed by the Connection Machine computer. This paper describes the first commercial implementation of Connection Machine architecture, the Thinking Machines Corporation Model CM-1.

Other computers have been proposed and built that have varying similarities to the CM-1 architecture. Some machines differ in their control such as the BBN Butterfly[1] and the NYU Ultracomputer[2]. Other machines differ in their inter-communication networks such as the Massively Parallel Processor[6], Non-Von[8], Dado[9], and ICL DAP[10]. Most of the ideas in the CM-1 system have their roots in the long history of both parallel and serial processors that have been proposed and built. The similarities and differences with these machines will not be covered in this paper.

Figure 1: The Connection Machine System


Processor/Memory/
Communication
I/O
Bus
VAX or
Symbolics
3600
FE
FE
FE
FE
N
e
x
u
s
S
w
i
t
c
h
16K processors
16K processors
16K processors
16K processors
Graphics
Display
Disk
Other High-speed
Interfaces

2 Hardware Overview

The CM-1 system consists of a processor array, from one to four front-end computers, and high speed peripherals such as disks and image devices (see Figure 1). The processor array contains 64K processing elements, each a simple serial processor with 4K bits of memory giving 32 MBytes of total memory for the machine. A full CM-1 system is made up of four *sections* of 16K processors each. The sections can be used separately, in pairs, or as one 64K processor unit.

Aside from its processors, each section contains inter-processor communication hardware,

a *sequencer*, and, optionally, a set of peripherals. The inter-processor commumication hardware is used for communication between any processors controlled by one host. A sequencer receives macro-instructions from the front end and broadcasts sequences of micro-instructions to all the processors in its section. A typical macro-instruction would be to add two 32-bit numbers and store them in a particular location. Because of the simplicity of the processors, each macro-instruction is typically implemented by many micro-instructions.

The front end computer, or host, attaches to the microcontroller through a bidirectional crossbar switch called a *Nexus*, and control one, two, or four 16K-processor sections. Typical front ends are a VAX®[2] or a Symbolics 3600®[3] Series computer.

Although the front end can be used to supply data to the CM-1 computer, in many applications the CM-1 processors can process data much faster than the front end can supply it. For this reason, the CM-1 processors are connected to a high speed bidirectional bus consisting of up to 8 I/O channels, each with a bandwidth of 320Mbits/sec. Special disk drives, frame buffers, frame grabbers, and specialized I/O devices can effectively use this bandwidth. This large I/O bandwidth makes the machine effective in very large, data-intensive problems such as databases, image analysis and graphics.

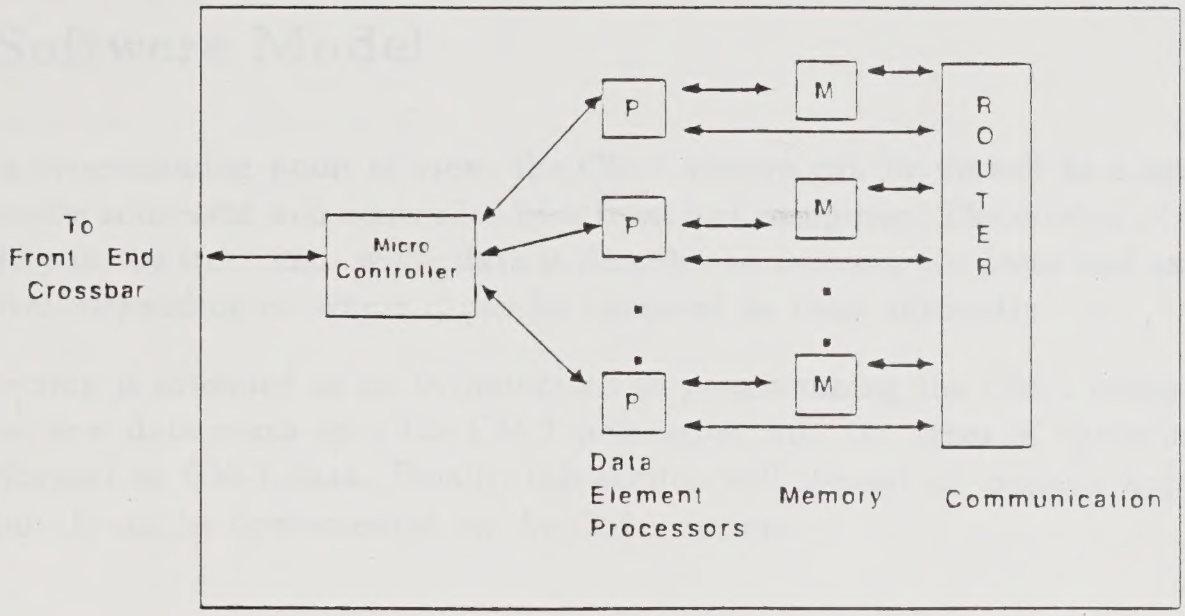
---

[2] VAX is a registered trademark of Digital Equipment Corporation

[3] Symbolics 3600 is a registered trademark of Symbolics, Inc.

Figure 2

Processor/Memory/Communication System



Pointer referencing (or referencing the data in one data object from another) on the CM-1 computer requires inter-processor communication. Since a CM-1 processor needs to be able to access data from any other processor, this inter-communication system has to handle a large load at high speed. This is achieved by use of a router which is integrated into the architecture so that every processor's memory is easily accessible to every other processor. The result is that the application programmer does not have to worry about physical processor geometry since the router handles all inter-processor communication efficiently regardless of layout.

Along with general pointer referencing, 2-dimensional inter-processor communication is supported. In this type of communication, each data object can communicate with its 2-dimensional neighbor (to the north, east, west, or south). This intercommunication is handled by a slightly different mechanism, and is faster than using the general communication mechanism. Image processing applications often use this ability to move data from one pixel to the next.

Performance of these individual elements will be treated later in this paper.

3 Software Model

From a programming point of view, the CM-1 system can be viewed as a smart memory that can be addressed and controlled by a front end computer. The control of the program lies solely in the front end, while data is distributed between the front end and the CM-1 computer, depending on where it can be operated on most efficiently.

This section is intended as an introduction to programming the CM-1 computer. It will describe how data maps onto the CM-1 processors, and the types of operations that can be performed on CM-1 data. Finally, this section will present an example application and show how it can be implemented on the CM-1 system.

3.1 The CM-1 Computer as Smart Memory

At the heart of any large problem is the data set consisting of some combination of interconnected data objects such as numbers, characters, records, structures, and arrays. In any application this data must be selected, combined and operated on. Data parallelism takes advantage of the parallelism inherent in large data sets.

To make effective use of data parallelism, the CM-1 computer has to be made to "look" like the data of the problem that is being solved. In order to do this, the processors in the machine are each assigned one data object, which can be any combination of immediate data and pointers. Operations can be specified to operate simultaneously on any or all data objects in the machine.

The CM-1 processors should be used whenever an operation can be performed simultaneously on many data objects. Data objects are left in the CM-1 computer during execution of the program, and are operated on in parallel at the command of the front end. This differs from the serial model of reading each data object from its memory one at a time, operating on it, and then re-storing it.

The front end computer handles the flow of control, storage and execution of the program,

and all interaction with the user and/or programmer. The data set, for the most part, is stored on the CM-1 computer. The front end can fetch and store memory in individual processors effeciently to perform serial operations if this is desirable.

There are several direct benefits to maintaining program control only on the front end. First, programmers can work in an environment which is familiar to them. Programming languages, programming concepts, and program development support (compilers, debuggers, editors, etc) do not change. Second, most of the existing code for an application does not have to be changed to use the CM-1 processors. A large part of most application programs pertains to the interface between the program, the user, and the operating system. Since the control of the program remains on the front end, code developed for these purposes is useful with or without the CM-1 computer, and only the code which pertains specifically to the data residing on the Connection Machine computer needs to be coded to use the CM-1 processors. Parts of the program which are especially suited for the front end, such as file manipulation, user interface, and serial data operations, can be done on the front end, while the parts of the program which run efficiently on the CM-1, namely the "inner loop" which operates on the data set, can be done there. In this way, the individual strengths of both the serial front end and Connection Machine computer can be maximized.

Most large problems have a large data set composed of interconnected structures, arrays, and records which can be mapped directly onto CM-1 processors. *Each processor's memory in the CM-1 computer holds the data for one record or array element and some small stack space for temporary results.* A processor can hold data of different types such as integers, floating point numbers, symbols, or pointers. Typically, each processor only needs 128 to 1024 bits for this information. Rarely is running out of memory with one processor a problem because each physical processor contains 4K bits of memory.

Problems of different sizes will use different numbers of processors. Some problems will allocate and de-allocate processors during the course of running the program. Because of the CM-1's flexibility this works well. If a program needs fewer than 64K processors, then some sit idle, ignoring the instructions that are being sent by the front end. If a program needs more assigned processors than 64K, then many more "virtual processors" are available, (see Section 6, Virtual Processor Concept).

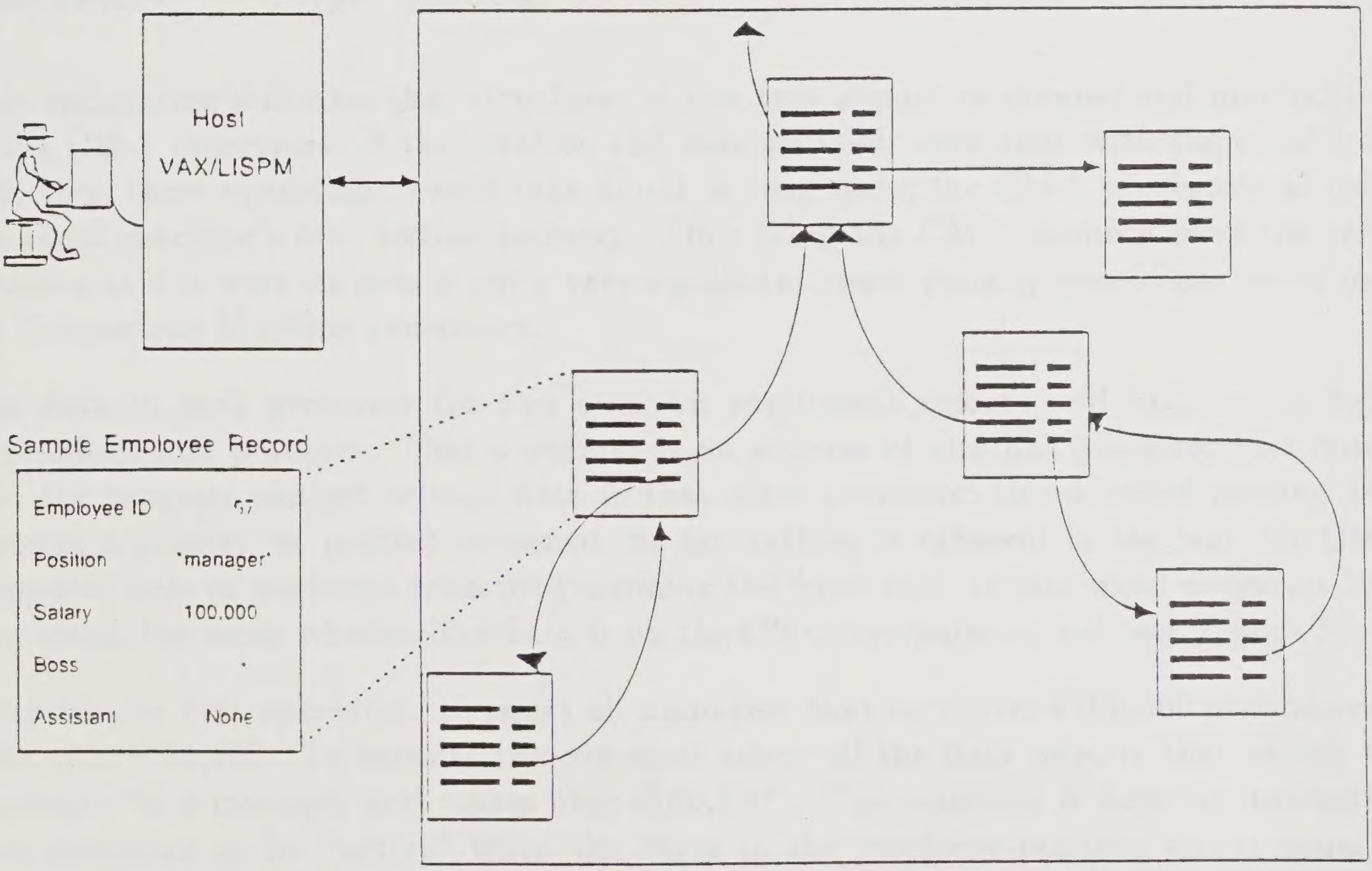
Examples of data objects used in the CM-1 computer are pixels, transistors, database records, atoms, sections of physical space. Each of these are allocated one per processor.

Throughout this paper the terms "processor", "data object" and "record", are used interchangeably.

3.2 Program Example

Consider the problem of manipulating a database of employee records where each employee is a structure of employee ID, title, salary, a pointer to their boss, and a pointer to their assistant (if they have one). The example operations are "give each manager that earns over $100,000 a pay cut of $5,000", "give all managers' assistants a 10% raise" and "sum all the managers' salaries" to illustrate selection and parallel operation, pointer reference, and global operations respectively.

Figure 3
Connection Machine Processors: Software View


Host
VAX/LISPM
Sample Employee Record
Employee ID
67
Position
'manager'
Salary
100,000
Boss
Assistant
None

For this sample problem, each employee's record would be assigned to a processor. On a serial machine data is loaded from a file and put into records in the virtual memory of the serial machine. On the CM-1 system, the difference is the records are kept on the CM-1 computer.

The language *Lisp[4] (pronounced star-lisp) is used in this paper to describe the pieces of code for the CM-1 computer. All examples can run on the CM-1 computer. In *Lisp the employee structure definition might look like:

```
(*defstruct (employee))
  employee-ID   ;type: number
  position      ;type: number
  salary        ;type: number
  boss          ;type: pointer
  assistant)    ;type: pointer
```

This declaration indicates that structures of this type should be created and manipulated in the CM-1 processors. If the creation and manipulation were done with the serial front end, then these operations would take about as long using the CM-1 processors as using the serial machine's own virtual memory. Thus using the CM-1 memory from the serial machine as if it were its own is not a very significant speed penalty even if one never used the Connection Machine processors.

The data in each processor (in this case, an employee's record) will be a set of fields of numbers and pointers. That a pointer is an address of another processor just means that the program can get or send data to that other processor; this is called moving data through a pointer, or pointer reference. So far nothing is different in the way the CM-1 computer feels or performs from programming the front end. In fact most programs look very much the same whether the data is on the CM-1 computer or not (see Figure 3).

Consider the first operation, to select all managers that earn over $100,000 and decrease their salary $5,000. To execute this we must select all the data objects that satisfy the predicate "is a manager and makes over $100,000". This selection is done by instructing each processor to be "active" when the value in the employee-position slot is equal to

---

[4] *Lisp is a trademark of Thinking Machines Corporation

`*manager*` and the value in the employee-salary slot is greater than $100,000. Notice that `*manager*` is a value known by the front end and is broadcast to each processor to compare against its local data. In *Lisp the front end code might look like this:

```
(*when (and (=!! position (!! *manager*))
            (>!! employee-salary 100000)))
  (*set employee-salary (-!! employee-salary (!! 5000)))
```

Executing this code on the front end would deliver an instruction stream that would look something like:

```
(let ((saved-context-bit *stack-index*))
  (CM:store-context-always saved-context-bit)
  (CM:u=constant (pvar-location position) *manager* 8)
  (CM:and-context-with-test)
  (CM:u>constant (pvar-location employee-salary) 100000 20)
  (CM:and-context-with-test)
  (CM:u-constant (pvar-location employee-salary) 5000 20)
  (CM:load-context-always saved-context-bit)
  )
```

Notice that these instructions look like a serial computer's macro-instruction set. We have selected a set of processors and executed an operation within those processors.

To perform the "give each managers' assistant a 10% raise" instruction requires communication between processors. This is done by referencing data in the processor pointed to by a field in another processor. In other words, a processor would have an address (pointer) in one of its slots to another processor.

This operation is done in the following steps:

- each manager's processor gets the salary information from their assistant (in the program this is "pref!!"; or parallel reference)

- the manager's processor saves that information in a temporary location. This is a move with the processor to the stack (the "*let")
- the new salary is computed by multiplying the old one by 1.10 in each manager's slot (the "*!!")
- the result is sent back to the assistant's processor and it's put in the salary slot. This involves another move through a pointer. (This is the "*pset", or memory parallel set)

  This operation is done, in *Lisp, as

```
(*let ((assistants-salary!! (pref!! assistant employee-salary)))
   (*pset assistant employee-salary
          (*!! assistant-salary!! (!! .10))))
```

The operation, "total all managers' salary", returns a number to the front end machine which is the total of all managers' salary slots. This is done by selecting all managers and then doing a *global sum.* Global sum is implemented using a logarithmic time tree reduction using the inter-communication wires directly. In *Lisp this looks like:

```
(*when (=!! position (!! *manager*)) ;select managers
  (*sum employee-salary))            ;returns to the front end the total
                                     ;of all processors employee-salaries.
```

This is an example of the use of *scan* that is discussed in the next section.

This example shows how a specific example might map onto the Connection Machine processors and how common operations are used on such data.

4 Basic Performance of the CM-1 System

The performance of the CM-1 system depends on the relative speeds of different common operations. Each application uses a different mix of instructions, so exact

application performance depends on processor utilization and the weighted average of the instructions used. Performance rates shown in Table 1 describe 32-bit operations from the point of view of an active processor as a ratio to the speed of an *add*. Further, this section describes how these numbers scale with different arithmetic precisions and "virtual processor" ratios.

Performance per processor is the operation speed of an active processor or data object. A 32-bit integer addition, for instance, takes about 32 microseconds per processor. The system performance depends on both the speed per processor and the number of active processors. Thus the peak system performance in additions is about 2000 MIPS (million instructions per second). Other elements which impact on performance are the instruction mix used by the application, the precision of the arithmetic and the total number of processors used. Some applications, such as image processing and cellular automata, run significantly faster than 2000 MIPS because they commonly use instructions on less than 32 bits. Table 1 uses 32 bits as a unit of measurement for other performances.

Table 1

Relative Times of Various 32-Bit Operations

Add	1
Move within Processor	.6
Multiply	32
Move with nearest grid neighbor	4
Move through pointer (pointer reference)	25
I/O move (out or into the CM-1)	20
Scan	25
Global max	2
Global sum	25

Table 1 describes the machine as a set of ratios with ADD time. Each entry is described here:

- *The 32-bit add* is the unit in Table 1. An add is a typical logical operation where the operands and results stay within each processor. Each of these fields might be on the stack or a slot in the data object. Other operations having similar speeds are logical operations such as OR, AND, XOR, etc. Also, if one of the operands to these instructions is an "immediate" number (a literal number that comes from the instruction stream rather than from local processor memory) the time is about the same.
- *Move within a processor.* This command moves a field from one location to another within each processor. An example of this is moving an argument to the stack.
- *Multiply* is implemented with a "shift-add" for each bit, so it takes 32 times longer than each add because it is a 32-bit multiply. This corresponds to the *!! in the program example.
- *Move with nearest grid neighbor* are transfers of 32 bits with the processor to the "east", "west", "north" and "south". All active processors will do this transfer at once. This operation is useful in image processing and graphics applications.
- *Move through a pointer* is a reference into another processor's memory. This corresponds to the pref!! (get the assistants' salaries) in the program example. The performance of this operation varies only slightly with the layout of the processors. Of course, if the problem is a grid problem, then the grid communications should be used instead.
- *I/O move* is the time required to get to the I/O port on the CM-1 computer. The real I/O performance will depend on the I/O device being used because most will not sustain a 320Mbit/sec/8K processor rate. The important aspect in this performance is its balance with the other performances so that the CM-1 computer can be connected directly into data-intensive applications.
- *Scan* is an operation that performs a form of processor intercommunication very quickly. Add-scan, for instance, results in each processor getting the sum in all processors before it. This is implemented with a parallel algorithm. Max-scan leaves the maximum up to that point.

Processor #	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	0	0	1	1	2	1	0	1
Result of Add-Scan	0	1	2	2	2	3	4	6	7	7	8
Result of Max-Scan	0	1	1	1	1	1	1	2	2	2	2

This is a useful operation on the CM-1 computer for implementing other instructions such as global-sum but many uses have little analogy with serial software cliches.

- *Global-max* results in the maximum value held by any active processor being passed back to the front end computer. The speed of this instruction is due to special hardware. Executing an `AND` of a field in all processors uses the same mechanism and runs at approximately the same speed. Similarly, `OR` and `MINIMUM` perform about the same.
- *Global-sum* is the sum of all values held by processors in the CM-1 computer. This is implemented using *add-scan* and taking the highest processor's result.

5 Performances with Different Word Lengths

The effective performance will change based on what precision arithmetic is needed. "Word Length" is used to describe the precision of the arguments to the CM-1 instructions. The previous section assumed 32-bit arithmetic for each of the operations. The Connection Machine system can operate on a few bits up to hundreds of bits with the same ease of programming. Variable length arguments are handled in microcode for efficiency and ease of programming. The language *Lisp, for example, can extend the word length of numbers as needed. Sometimes a program will need only 16-bit numbers to start, but will use 32 or 64 bit numbers later (or 53 bits for that matter). It is, of course, desirable to minimize the precision required to maximize performance and minimize space utilization.

As one can see in Table 2, most operations scale linearly with the word length. Thus if only 16 bit additions are needed, twice as many can be done in the time of a 32-bit addition. There is a small amount of calling overhead in each instruction call so behavior is not quite linear for very short numbers.

Table 2

Relative Times for Operations on N-bit Numbers

Add	N
Move within Processor	N
Multiply	N^2
Move with nearest grid neighbor	N
Move through pointer (pointer reference)	N
I/O move (out or into the CM-1)	N
Scan	N
Global max	N
Global sum	N

Multiplication uses a shift-add algorithm so that the precision scales with the square of the number of bits; thus a 16 bit multiply would take 1/4 the time of a 32-bit multiply.

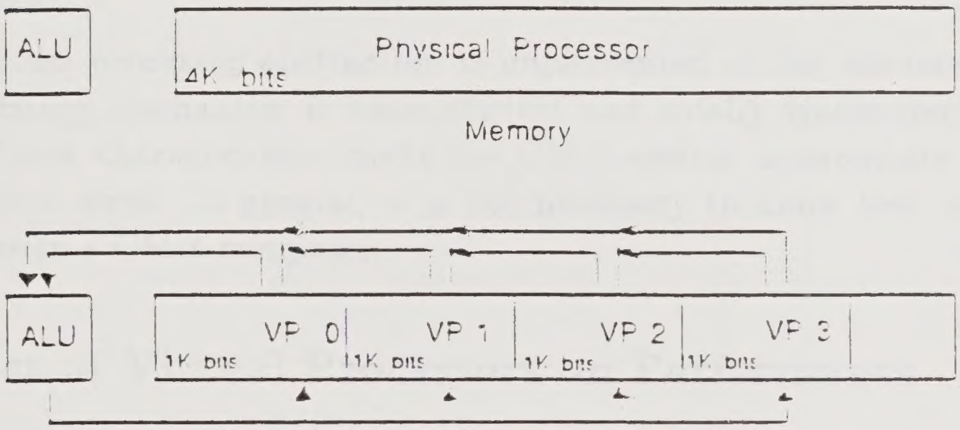
The CM-1 system's flexibility extends to the precision of the arithmetic needed. Image processing applications often use very limited precision numbers, and numerical simulations often use very large precision numbers.

6 Virtual Processor Concept

As presented so far, the CM-1 system has a constant (independent of problem size) performance for an add operation for problem sizes up to 64K processors. In real applications, often 64K processors are not enough. For example, an image processing application with a 1000 by 1000 image would require 1 million processors, one for each pixel. In order to get around the arbitrary limit of the number of physical processors, the CM-1 supports *virtual processors*. Virtual processors are a software abstraction, implemented at the microcode level, which allows a programmer to write programs which are independent of the number of physical processors that the CM-1 hardware contains. This section will describe how virtual processors are implemented on the CM-1, and what effect they have on program performance.

Figure 4

Virtual Processor (VP) Concept



Virtual processors are implemented using three separate mechanisms; one for storage, one for processing, and another for communications. First, the memory of each physical processor is divided evenly among the virtual processors assigned to it. The number of virtual processors per physical processor is referred to as the *virtual processor ratio*. For example, a problem requiring 1M (2^{20}) virtual processors would need a virtual processor ratio of 16 to operate on a 64K machine ($2^{20}/2^{16} = 2^4 = 16$). This means that the physical address space of each virtual processor would be divided into 16 equal address spaces, one for each virtual processor assigned to it. Image processing applications commonly use only 100 bits per virtual processor so

32 processors can be assigned to each physical processor, and the CM-1 computer can be configured as a 2 million processor machine.

The second mechanism which is necessary to support virtual processors is time-multiplexing of the physical processors among the virtual processors assigned to it. Every macro-instruction that is sent by the front end is run on each of the virtual processors within each physical processor. The overhead for switching context is extremely small (about the time to execute a 2-bit add) because each processor is so simple. In the 1M processor example given above, each macro instruction would be executed 16 times by each physical processor.

The third mechanism of communications allow the CM-1 processors to communicate with one another without regard to virtual processors. Grid communications are handled by the microcode by sharing the grid wires. General communications (pointer reference) is handled by the router hardware. The length of a processor address changes based on the number of virtual processors in the machine. This virtual address is used by the router hardware to deliver messages to the correct virtual processor.

Since the virtual processor abstraction is implemented at the microcode level, the virtual processing mechanism is both efficient and totally transparent to the programmer. These characteristics make the CM-1 system appropriate for problems of very different sizes. In general, it is not necessary to know how many physical processors are in a CM-1 computer.

6.1 Effect of Virtual Processors on Performance

Since a physical processor is shared among several virtual processors, the performance from a virtual processor's point of view goes down. If one measures the performance of the CM-1 computer as a whole, however, in terms of MIPs or MFLOPs, the performance does not go down but remains constant. In other words, given a virtual processor ratio of 16, each virtual processor is only getting 1/16 of the physical processors time, so that each macro-instruction takes 16 times as long, but each macro-instruction is doing 16 times as much work as in the case of a virtual processor ratio of 1.

Table 3 shows the effects of virtual processors on the performance of the various classes of macro-instructions. As expected, most operations slow down linearly with

Table 3

Relative Time of Operation with K Virtual Processors

Add	K
Move within Processor	K
Multiply	K
Move with nearest grid neighbor	K
Move through pointer (pointer reference)	KlogK
I/O move (out or into the CM-1)	K
Scan	K
Global max	K
Global sum	K

the virtual processor ratio. The only exceptions are those that involve interprocessor communication which slow down by $O(Klog(K))$ for virtual processor ratios less than 16. The performance for very high ratios has not been studied. The addition of the $log(K)$ term is due to the increase in router congestion.

Virtual processors free the programmer from the physical size of the CM-1, and also from the size of the data set that the program is to operate on. Since this is done at a very low cost in speed and programming, virtual processors have been used in most applications programmed for the CM-1 system.

7 Conclusion

The Model CM-1 system offers a high level of performance due to its flexible, massively parallel architecture. Much of the maximum peak performance of 2000 MIPS

can be used on a problem due to the balanced performances of all common instructions. The CM-1 can be used for very different types of applications because the machine can be efficiently reconfigured to have the needed number of processors and the needed arithmetic precision.

Since the CM-1 computer works closely with a serial front end computer, the programmer operates in a familiar environment. The CM-1 system combines the benefits of programming the front end computer with the performance of a massively parallel machine.

References

[1] Bolt Beranek and Newman Inc. "Development of a Butterfly Multiprocessor Test Bed," Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[2] Gottlieb, Allan, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. "The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, Vol. C-32, No. 2, February 1983, pp. 175-189.

[3] Hillis, W. Daniel, *The Connection Machine*, The MIT Press, Cambridge, MA 1985

[4] Introduction to Data Level Parallelism, Thinking Machines Technical Report, 86.14, April 1986

[5] Lasser, C. and S. Omohundro, "The Essential *Lisp Manual", Thinking Machines Corporation, July 1986

[6] Schaefer, D. H., J. R. Fischer, and K. R. Wallgren. "The Massively Parallel Processor," *Engineering Notes*, Vol. 5, No. 3, pp. 313-315, May-June 1982. (More MPP)

[7] Siegel, Howard Jay, Leah J. Siegel, Frederick C. Kemmerer, Philip T. Mueller, Jr., Harold E. Smalley, Jr., and S. Diane Smith. "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, Vol C-30, No. 12, December 1981.

[8] Shaw, David Elliot. "The NON-VON Supercomputer," Department of Computer Science, Columbia University, August 1982.

[9] Stolfo, Salvatore J. and David Elliot Shaw. "DADO: A Tree-Structured Machine Architecture for Production Systems," Department of Computer Science, Columbia University, March, 1982.

[10] Flanders, P.M., et. al. "Efficient High Speed Computing with the Distributed Array Processor," *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. Academic Press, New York, 1977, pp. 113-127.

Thinking Machines Corporation **The Connection Machine Supercomputer**

PRODUCT SPECIFICATIONS

Processors	4096	8192	16,384	32,768	65,536
Memory	128-512 MBytes	256 MBytes-1 GByte	512 MBytes-2 GBytes	1 GByte-4 GBytes	2 GBytes-8 GBytes
Performance (Processor)					
32-bit integer add	150 Mips	300 Mips	600 Mips	1200 Mips	2500 Mips
8-bit byte add	250 Mips	500 Mips	1000 Mips	2000 Mips	4000 Mips
4K x 4K Matrix Multiply					
32-bit IEEE floating point	310 MFlops (2K x 2K)	625 MFlops	1250 MFlops	2500 MFlops	5000 MFlops
64-bit IEEE floating point	–	–	625 MFlops	1250 MFlops	2500 MFlops
Dot Product					
32-bit IEEE floating point	625 MFlops	1250 MFlops	2500 MFlops	5000 MFlops	10,000 MFlops
64-bit IEEE floating point	–	–	1250 MFlops	2500 MFlops	5000 MFlops
Performance (Example Applications)					
Poisson's Equation (32-bit)	230 MFlops	475 MFlops	950 MFlops	1900 MFlops	3800 MFlops
Wave Equation (32-bit)	875 MFlops	1750 MFlops	3500 MFlops	7000 MFlops	14,000 MFlops
Languages	*Lisp, C*	*Lisp, C*	*Lisp, C*	*Lisp, C*	*Lisp, C*
	Fortran	Fortran	Fortran	Fortran	Fortran
Front End Systems	Sun 4, VAX	Sun 4, VAX	Sun 4, VAX	Sun 4, VAX	Sun 4, VAX
Input/Output Channels					
Number of Channels	2	2	4	8	8
Capacity per channel	50 MBytes/sec	50 MBytes/sec	50 MBytes/sec	50 MBytes/sec	50 MBytes/sec
Maximum Transfer Rate	50 MBytes/sec	50 MBytes/sec	100 MBytes/sec	200 MBytes/sec	400 MBytes/sec
Environmental Requirements					
Power	30 amp 1-phase	30 amp 1-phase	30 amp 3-phase	30 amp 3-phase	30 amp 3-phase
Voltage	200-240	200-240	110/208Y	110/208Y	110/208Y
Hertz	50/60	50/60	50/60	50/60	50/60
Power Dissipation	3.5 KVA	3.5 KVA	7 KVA	14 KVA	28 KVA
Size	26″ x 26″ x 43″	26″ x 26″ x 43″	56″ x 56″ x 62″	56″ x 56″ x 62″	56″ x 56″ x 62″
Weight	375 lbs.	375 lbs.	1200 lbs.	1800 lbs.	3000 lbs.
Operating Temperature	70°F ± 5°F	70°F ± 5°F	70°F ± 5°F	70°F ± 5°F	70°F ± 5°F
Relative Humidity	50% ± 10%	50% ± 10%	50% ± 10%	50% ± 10%	50% ± 10%

Connection Machine is a registered trademark of Thinking Machines Corporation.
DataVault is a trademark of Thinking Machines Corporation.
C* and *Lisp are trademarks of Thinking Machines Corporation.

VAX is a trademark of Digital Equipment Corporation.
Sun is a trademark of Sun Microsystems, Inc.
Symbolics 3600 is a trademark of Symbolics Incorporated.

Thinking Machines Corporation believes all specifications are accurate as of the date of publication. Thinking Machines cannot, however, be responsible for inadvertent errors. Product specifications subject to change without notice.

© Copyright 1991 Thinking Machines Corporation.

A new implementation of free-text search using a new parallel computer—the Connection Machine®—makes possible the application of exhaustive methods not previously feasible for large databases.

PARALLEL FREE-TEXT SEARCH ON THE CONNECTION MACHINE SYSTEM

CRAIG STANFILL and BREWSTER KAHLE

Locating information in natural-language documents contained in large databases is an important problem. Such databases may contain articles from wire services and newspapers, abstracts and full-text articles from journals and encyclopedias, as well as bibliographies and judicial rulings. Search facilities for such databases are provided by vendors like Dialog, Lexis, Nexis, Westlaw, Medlars, and VU-Text. In many communities, particularly legal and biomedical research, access to such databases has become critical. The size of such databases can be virtually unlimited: The Lexis database, for example, reportedly contains a total of 125 Gbytes of information.

The two general methods of accessing such databases are free-text search and keyword search: Free-text search systems allow the user to search the text of the documents for arbitrary combinations of words, whereas keyword systems require that indexers read all the documents in a database and assign them keywords from a controlled vocabulary. In the latter system, the user searches for information or documents using some combination of keywords from this controlled vocabulary. The relative merits of these systems continue to be debated [1, 11]. Advocates of manual indexing systems argue that free-text search systems suffer from poor recall (they fail to retrieve all relevant documents) and poor precision (they deliver too many irrelevant documents). Proponents of free-text search counter that techniques are available to alleviate these two problems and that manual indexing systems have problems of their own: specifically, that the manual indexing process is unreliable, based on the observation that different indexers will often assign radically different keywords to the same article.

Although techniques are available to improve the recall and precision of searches in free-text database systems, they are in some cases not widely used because they make computational demands beyond the practical limits of conventional computers.

The Connection Machine is a registered trademark of Thinking Machines Corporation.

© 1986 ACM 0001-0782/86/1200-1229 75¢

The usual method of organizing a free-text database is to construct an inverted index of all words it contains [4, 12]. To locate the documents containing a given word, the index for that word is located (via hashing, for example) and retrieved. Documents containing combinations of words are found by computing the intersection and union of indexes. However, inverting a database in this way often increases the storage utilization 200 percent over the raw text.

This article will take no stand on the issue of the relative merits of free-text versus controlled-vocabulary searching. Rather, we will present a new implementation of free-text searching that takes advantage of the possibilities offered by a massively parallel computer, the Connection Machine (CM), with up to 65,536 processing elements [3]. In this implementation, the representation of the documents in memory permits a very fast search for the presence of a word in a document. With the CM, each processing element stores between one and three documents; queries are then broadcast to the entire machine and the results collected. Because of the massive parallelism, the resulting system is fast enough to permit the application of exhaustive methods not previously feasible for large databases.

Two applications of this technology are discussed: The first is a benchmark of the CM as a query evaluator. In this application, a Boolean query is evaluated against a sample database. Using this measurement and taking into account the approximate performance of the I/O system, we estimate that the CM can evaluate a 20,000-term query against a 15 Gbyte database in 3 minutes, of which 2 minutes are devoted to I/O and 1 minute to computation. The second application, a prototype of an interactive document-retrieval system, uses a technique called relevance feedback to produce an interface that is fast and easy to use, and produces a high-quality search.

THE CONNECTION MACHINE SYSTEM

The Connection Machine System is a fine-grained, highly parallel computer whose current version consists of between one and four modules of 16,384 processing elements each, for a maximum configuration of 65,536 processing elements. Each of these elements has 4,096 bits of memory and a 1-bit-wide ALU (arithmetic and logical unit), so that adding two 32-bit numbers takes 32 machine cycles. Each module is controlled by a single microcontroller, which executes macroinstructions originating in a host computer. The modules can be operated in concert, running one program, or independently, running up to four different programs. A machine cycle (3 microcontroller states) takes about 750 ns, yielding a raw system throughput exceeding two billion 32-bit operations per second. When operating on data smaller than 32 bits, proportionally higher processing rates are obtained. For the document-retrieval system about to be described, the measured performance is approximately six billion operations per second.

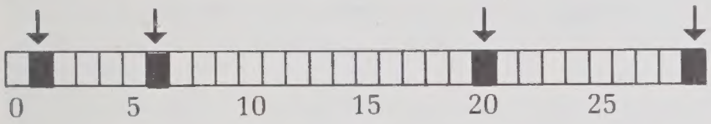
All processing elements execute the same program, with each operating on the contents of its own memory. The ALU has a context flag that allows processors to be selectively disabled. Communication is provided by three independent communication networks: a 1-bit-wide global-OR network, a 2-dimensional grid, and a 12-dimensional hypercube with 16 processors at each vertex. A global-OR network allows such operations as global-minimum, global-maximum, and global-OR to be performed in a time span proportional to the length of the operands: on the order of microseconds per bit. The grid allows instantaneous communication between nearest neighbors. The hypercube network supports full packet-switched communication between arbitrary processors. A disk system, which will allow the CM's full memory capacity of 32 Mbytes to be swapped in and out of mass storage in about 0.6 s, is currently under design. With the exception of the hypercube network and the disk system, the CM is similar to the Massively Parallel Processor (MPP) [9].

DATA STRUCTURE

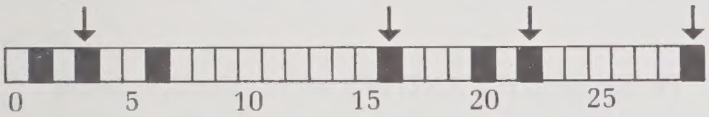
The starting point of the information-retrieval system is the representation of documents in the CM's memory. The basic requirement is to be able to very quickly test the machine's memory for the presence of a word. In addition, the representation must be reasonably compact. We selected a representation that was originally developed for spelling-correction dictionaries and is known as *surrogate coding* [2, 7, 8]. The advantage of surrogate coding is that probing a surrogate table for the presence of a word requires only AND-ing together a small number of bits; the disadvantages are that word order and proximity information are lost, and probes sometimes return false positives.

A surrogate table is a binary array k bits long. In the example that follows, we will use $k = 30$, although in actual implementations we have used values of 512 and 1024. The table is initialized to all zeros.

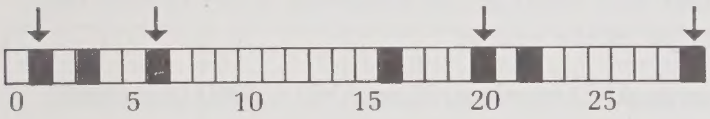
To insert a document into the table, i independent hash codes between 0 and $k - 1$ are generated. Here, we will use $i = 4$, although, in practice, i is usually between 10 and 30. If we insert the word "chemistry" into an empty table, and *hash*(*chemistry*) = (1, 6, 29, 20), setting these four bits to 1 produces the following table:



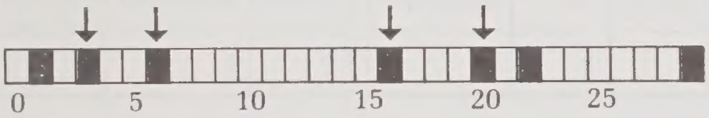
If we then insert a second word "biology" into the table, with *hash*(*biology*) = (16, 22, 29, 3), setting these four bits in the table produces the following layout. Note that one of these bits (29) was already set.



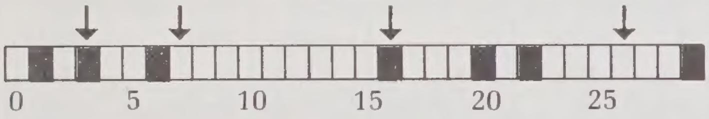
To probe for a word, we generate its *i* hash codes and then AND together the corresponding bits. To probe for "chemistry," we AND together bits 1, 6, 29, and 20. Since all four are 1, the probe returns PRESENT.



Suppose we probe for a second word, "physics," which was not inserted in the table. Assuming that *hash*(*physics*) = (3, 7, 16, 26), we check the table and find that bits 7 and 26 are 0. Thus, the probe returns ABSENT.



It is possible for a probe to return a false positive. In the case of *hash*(*mathematics*) = (3, 6, 16, 20), checking those four bits will show that all are 1. Thus, the probe returns PRESENT, even though "mathematics" was never inserted in the table.



The probability of a false positive depends on the size of the table (k), the number of words encoded in the table (w), and the number of bits that are set for each word (i). Decreasing w reduces the false-hit rate, but also increases storage needs. Increasing i, up to a point, also reduces the false-hit rate, but increases processing time. An analysis that will provide some guidance in making the most appropriate trade-off is presented next.

Analysis of the Data Structure

The probability of a false positive may be determined by finding the probability that a random bit is 1 as a function of k, w, and i, and then using this result to calculate the probability of i random bits all being 1. Given w words, and i bits set for each word, a total of $w \times i$ bits will be set. We will now calculate $P_1(w, i, k)$, the probability that a random bit in the table will be set. With a k-bit table T and a sequence S of $w \times i$ random integers uniformly distributed between 0 and $k - 1$, an element T_j of T is 1 if and only if j occurs in S; otherwise, it is 0. Therefore,

$$P_1(w, i, k) = 1 - P_0(w, i, k)$$

where $P_0(w, i, k)$ is the probability that a random number j, uniformly distributed between 1 and $k - 1$, does not occur in the sequence S. This gives us the following two formulations:

$$P_0(w, i, k) = \left(\frac{k-1}{k}\right)^{wi}$$

$$P_1(w, i, k) = 1 - \left(\frac{k-1}{k}\right)^{wi}$$

We now need to determine the probability that i random bits are all 1: We will call this $P_{false}(w, i, k)$. This is the probability of a false hit: That is, the probability that a word not present in the document used to create the table will return PRESENT when it is probed for.[1]

$$P_{false}(w, i, k) = P_1(w, i, k)^i = \left(1 - \left(\frac{k-1}{k}\right)^{wi}\right)^i$$

Table I shows some values of P_{false}, assuming $k = 512$ and $i = 10$. The important fact to remember about Table I (p. 1232) is that a small change in w causes a large change in P_{false}: That is, the false-hit rate may be dramatically reduced by a small decrease in the number of words in each table. Depending on the false-hit rate that is considered acceptable in any particular application, the values of w, i, and k can be tuned to yield optimal system performance.

APPLICATION

In practice, the application of this algorithm is slightly more complex than suggested above. First, each processor in the CM contains 4096 bits of memory, some of which are used as scratch memory. This allows six tables of 512 bits, or three tables of

---

[1] This calculation makes the simplifying assumption that the result of probing different locations in the table is statistically independent. This is not strictly correct, but for useful values of w, i, and k, it is not a bad assumption. Tom Kraay [5] has derived an exact form of P_{false} and shown it to be very close to our form for nonsmall values of w and k.

1024 bits. Moreover, as the above discussion indicates, it is necessary to limit the number of words in each table. For a 512-bit table, a limit of 15 to 30 words is reasonable. If a document contains more than the allotted number of words, several tables must be used for each document. Thus, if 30 words are being put into each table and a 90-word document is encountered, it is necessary to use three tables, which are generally put in three different processors. Such a set of tables is called a *chain*, where each chain reports to a designated *head processor*.

Probes for words are never used in isolation, but rather as components of Boolean queries (where they are combined via Boolean connectives), or as components of simple queries (where each word is given a numerical weight).

Boolean Queries

Boolean queries are built from primitive word tests that are combined via the Boolean connectives AND, OR, and NOT. To compute the OR of a set of word tests, each table is probed independently and the results for each table OR-ed together. These results are then passed on to the head processor of the chain and OR-ed together. If we had two documents, the first containing the words "A few words that might be in some document" and the second containing the words "Some other words that a document might well contain," and each document were broken up into three tables, considering the query

might OR right

would involve first probing each table for both words. Then, the process of OR-ing together the results of the probes would produce the following results:

A few words	*that might be*	*in some document*
absent	present	absent

Some other words	*that a document*	*might well contain*
absent	absent	absent

Passing the results to the head of each chain and OR-ing them together yield a hit on the first document.

A few words	*that might be*	*in some document*
present		

Some other words	*that a document*	*might well contain*
absent		

TABLE I. Probability of a False Hit

w = Words per table
i = Bits set for each word, fixed at 10
k = Bits per table, fixed at 512
P_{false} = Probability of a false positive being returned when probing one table for one word

w	P_{false}
5	4.91×10^{-11}
10	3.12×10^{-8}
15	1.13×10^{-6}
20	1.26×10^{-5}
25	7.46×10^{-5}
30	2.96×10^{-4}

To compute the AND of a set of primitive word tests, each table is probed independently, and the number of probes that return PRESENT is recorded for each table. The results are then passed to the head processor of the chain and summed. If the total is equal to the total number of terms in the AND clause, the clause returns true. This method requires that each word in the document occur in only one table. In the query

some AND words AND in AND a AND document

we first probe for each word, counting the number present in each table:

A few words	*that might be*	*in some document*
2	0	3

Some other words	*that a document*	*might well contain*
2	2	0

Sending the results to the head processor and adding reveal that the first document had $^5/_5$ of the words and therefore satisfies the query, whereas the second document had only $^4/_5$ and therefore fails the test.

A few words	*that might be*	*in some document*
5		

Some other words	*that a document*	*might well contain*
4		

The set of documents matching a particular query is extracted from the database using a global-maximum operation. Each document has a document number that is stored in its head processor. The query is executed, and processing elements that do not record a hit are masked out. The global-maximum operation is then used to find the largest numbered document in an active processor. That processor is then masked out, and the process continues until all selected documents have been extracted.

Simple Queries

In simple queries, point values are assigned to a set of words, and each document in the database is then scored by totaling the scores of the words it contains. This is done by adding up the score for the words in each processor and then totaling the scores of the processors in each chain. With the following simple query

some	2
words	2
in	1
a	1
document	4

we first probe each table for each of the five words, summing the scores of the words that are found:

A few words	*that might be*	*in some document*
3	0	7

Some other words	*that a document*	*might well contain*
4	5	0

We then add the scores for each table in a chain, yielding a score of 10 for the first document and 9 for the second.

A few words	*that might be*	*in some document*
10		

Some other words	*that a document*	*might well contain*
9		

The documents in the database that are assigned nonzero scores are retrieved best-to-worst. The global-maximum operation is used to find the processor in the CM with the highest total score; that processor is masked out, and the process repeated to extract the second highest score, and so forth. The process can be halted when a specified number of documents has been retrieved, or when only documents with a score below a specified threshold remain in the machine.

A BENCHMARK

In September 1985, a benchmark of the Boolean query algorithm was performed on a 16,384-processor prototype of the CM. The test database consisted of 31,994 documents, totaling 18 Mbytes. A document consisted of several fixed-format numerical fields, totaling 100 bits, and two variable-length text fields. Each virtual processor in the CM had two tables of 512 bits each, and the packing density was limited to 35 words per table. Several trial queries, containing between 25 and 20,000 atomic terms, were evaluated. The times required to execute the queries varied between 0.004 s for a 25-term query and 0.295 s for a 20,000-term query. For a full-sized CM with 65,536 processors, execution times would remain the same, while the database could grow to 128,000 documents totaling 71 Mbytes.

Databases larger than 71 Mbytes will require the use of the swapping disk to page additional segments of the database into the CM's memory. Swapping in the data should require approximately 0.6 s per database segment. From then on, evaluating a query again takes between 0.004 s (25 terms) and 0.295 s (20,000 terms) per database segment. Using these figures, the time required to search a 15-Gbyte database should vary between 2 min. for a 25-term query and 3 min. for a 20,000-term query—a substantial improvement over existing search systems. The results of this benchmark are summarized in Table II (p. 1234).

SEARCH STRATEGIES

To date, the methods used to search large databases have been limited by the nature of the sequential computers available to perform them. Since the basic problem is that an exhaustive search may take a very long time, on-line searches on sequential machines have traditionally required nonexhaustive methods such as searching inverted indexes. Although inverted indexes suffice for short queries, as the number of terms in the queries grows they become impractical. In particular, as the number of search terms becomes large, the number of documents having at least one of the terms approaches the size of the database itself. The CM's ability to conduct an exhaustive parallel search in a small amount of time changes the situation radically. Search strategies that have been discussed for some time, but not commercialized, have suddenly become practical.

In evaluating search strategy performance, we use three standard criteria. The first is a pair of quantitative parameters that together measure the quality of the search: *precision*, which is the proportion of retrieved documents that are relevant; and *recall*, which is the proportion of relevant documents in the entire database that are retrieved. The second criterion, *ease of use*, involves a determination of how quickly a user can learn to use the system, and how

TABLE II. Results of the Boolean Query Benchmark

Measured benchmark and projected times					
	Query size				
	25 terms	100 terms	1,000 terms	10,000 terms	20,000 terms
Load, one batch (s)	0.567	0.567	0.567	0.567	0.567
Execute query, one batch (s)	0.004	0.005	0.022	0.149	0.295
Total time per batch (s)	0.571	0.572	0.589	0.716	0.862
Hits per batch	22	22	25	50	68
Total time for 15 Gbytes (min.)	2.0	2.1	2.1	2.5	3.0

	Benchmark batch size (16K processors)	Full batch size (64K processors)	Full database size (211 batches)
Raw data	18 Mbytes	71 Mbytes	15 Gbytes
Compressed data	8 Mbytes	32 Mbytes	7 Gbytes
Documents	31,994	128,000	27,000,000

easy the system is to use once it has been mastered. Finally, there is *response time*: If a search system is too slow, it cannot be used for interactive search.

Boolean Queries

Although Boolean queries are typically composed of word tests plus the connectives AND, OR, and NOT, many variations on Boolean queries can be implemented by adding additional operators. Boolean queries implemented with inverted indexes are currently the primary method for searching large full-text databases. They are limited in the quality of the search they provide, and users find them clumsy, but they do deliver adequate performance.

However, there is a trade-off between recall and precision that limits the quality of Boolean searches on large databases [1]. Searching a database for documents containing a single word may lead to low recall because there is no guarantee that all relevant documents will use that word. In addition, it is likely that a large number of irrelevant documents will be retrieved, leading to low precision. Searching for several words together aggravates one or the other of these problems: If the searcher looks for any of several words (a disjunctive query), recall improves but precision goes down; on the other hand, if the searcher looks for documents containing all of several words (a conjunctive query), precision improves but recall suffers. For a large database, this means that the searcher may have to choose between missing important information and reading thousands of irrelevant documents.

There are other problems with the use of Boolean queries for full-text search. First, the user is playing a guessing game when trying to anticipate which words the authors of the documents he or she is interested in might have used. Second, even if the user guesses the words, he or she has to figure out which connectives to use to avoid getting either too much or too little data; this often involves several iterations as the query is debugged. Moreover, the syntax of Boolean queries is complex, making the systems difficult to learn.

As far as performance is concerned, Boolean search systems employing inverted indexes are fast enough to be commercially successful. Nonetheless, the problems with search quality and usability are severe, and there is wide agreement that something better is needed [1].

Simple Queries

An alternative to a Boolean search strategy is to use simple queries [12]. A simple query consists of a set of words, each of which is assigned a point value. Every document in the database is scored (for potential retrieval value) by adding up the point values for the words it contains. The total score can then be normalized to take into account the size of the document and the length of the query.

Simple queries are easier to use than Boolean queries because the user does not need to decide which connectives to use, as there are none, and it is not necessary to learn a complex query language, as the query consists of a list of words. However, the user does still need to guess which words to use in the query.

Relevance Feedback

Relevance feedback ([10, 12]) constructs simple queries from the texts of documents that have already been judged relevant. After some search method has been used to locate a small set of possibly relevant documents, the user scans these documents and marks any documents that are obviously relevant as *good*, and any that are obviously irrelevant as *bad*. The text of the marked documents is then scanned for words, and a simple query is constructed from these words. The more good documents a word occurs in, the higher its score.

Since simple queries built using this technique may contain hundreds of terms, its practical application on large databases will require a very powerful machine. Applying the CM to this task, since it has both sufficient processing power and sufficient I/O bandwidth, results in an excellent search in terms of quality, ease of use, and performance, as discussed in detail below.

THE SEED DOCUMENT SEARCH SYSTEM

The combined use of simple queries and the relevance feedback technique was tested in a prototype system running on the CM. The test database consists of approximately 16,000 articles from the Reuters wire service totaling approximately 32 Mbytes and dating from December 17, 1985, to February 1, 1986. In this test system, the text is run through the Context® Text Indexer [6], which does simple linguistic and semantic analysis to determine which words may be dropped from the document without affecting its retrievability. All words not dropped by the Indexer are encoded in the surrogate tables (described on p. 1232) and stored in a 16,384 processing element version of the CM that has 4,096 bits of storage per element. The interface is mouse driven and runs on a Symbolics 3600-series® Lisp machine. The full text of the data is stored on the Lisp machine's disk system.

To begin the search, the user types in a set of *seed words*. The search system assigns these words scores as a function of their frequency in the database as a whole, and evaluates the resulting simple query. Evaluating this simple query and retrieving the results takes approximately 20 ms. The results of this initial query are displayed as a list of citations[2] that the user can browse through by clicking a mouse. For example, to look for reports of former Philippine President Ferdinand Marcos's hidden assets in the United States, one might enter the seed words "`marcos hidden wealth`," which would produce the set of articles shown in Figure 1 (p. 1236).

During the process of perusing the initial set of retrieved documents, the user begins a second phase of search. When a document from this set is determined to be relevant to the topic, the user clicks a mouse on a box labeled `Good`. The search system notes the words contained in that document and assigns them scores for use in the next round of search. The user may mark as many documents as desired. When the search system is invoked again, all the words in the documents that the user has marked `Good` go into the simple query. The user may also mark documents `Bad`, which causes words that are in *bad* documents but not in *good* ones to be assigned inhibitory weight. This method yields a query composed of several hundred words. The results of such a search are typically available in approximately 1 s, which is largely consumed by overhead in the interface and support system; a search that generates 200 terms and retrieves 20 candidate documents uses only 60 ms of CM time. The results of this second phase (or *seed document*) search are usually much better in terms of higher recall and better precision than those attained in the first phase. In our example (Figure 1), we mark the document entitled "`MARCOS PROPERTY ON SALE FOR $300 MILLION, NEW YORK TIMES SAYS`" as a seed document (i.e., `Good`) and perform a second search that yields the set of documents shown in Figure 2 (p. 1237). The user may repeat the process as often as desired, with search results becoming better and better as the query is refined.

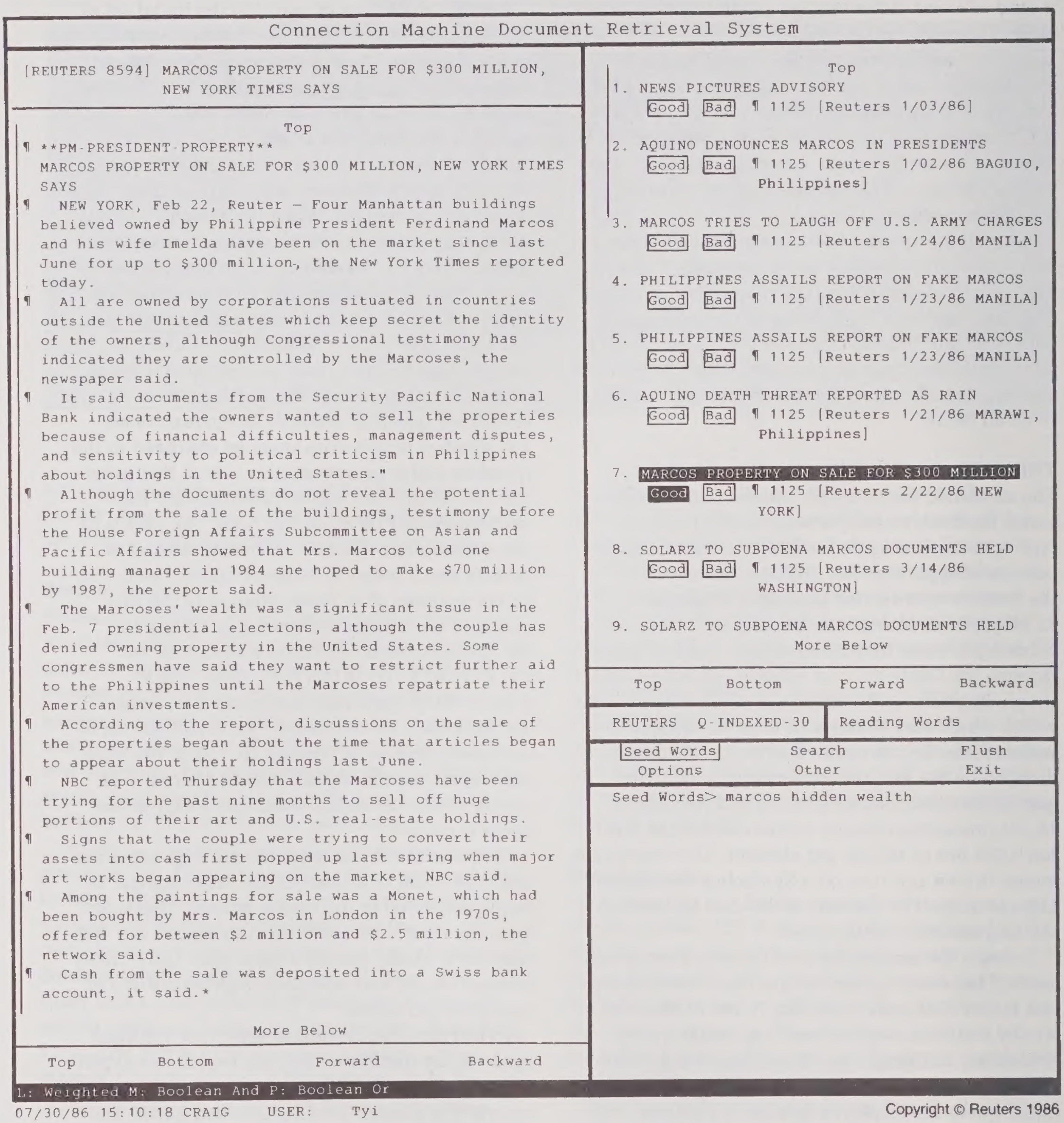
Relevance feedback leads to both high precision and high recall as a result of the large number of words employed in the search process. One word, taken by itself, conveys little information, but 200, taken as a whole, convey a great deal. Only highly relevant documents will use a high proportion of this set of 200 terms.

To test the effectiveness of relevance feedback vis-à-vis the traditional Boolean search, we experimented with two separate retrieval tasks: locating reports of a summit meeting between Jordan and Syria in late December 1985, and retrieving reports that Ferdinand Marcos had hidden some of his assets in the United States (our sample search shown in Figures 1 and 2). First, a very broad Boolean query was executed to search the database: In the first case, this query consisted of the word "Syria;" in the second, it was "Marcos." These two queries retrieved 288 and 218 documents, respectively. These sets of candidate documents were then browsed through to find relevant documents (14 and

---

[2] For the Reuters news database, the citations include headlines plus dates and places of filing.

Context is a trademark of Thinking Machines Corporation.
Symbolics 3600 is a trademark of Symbolics, Inc.



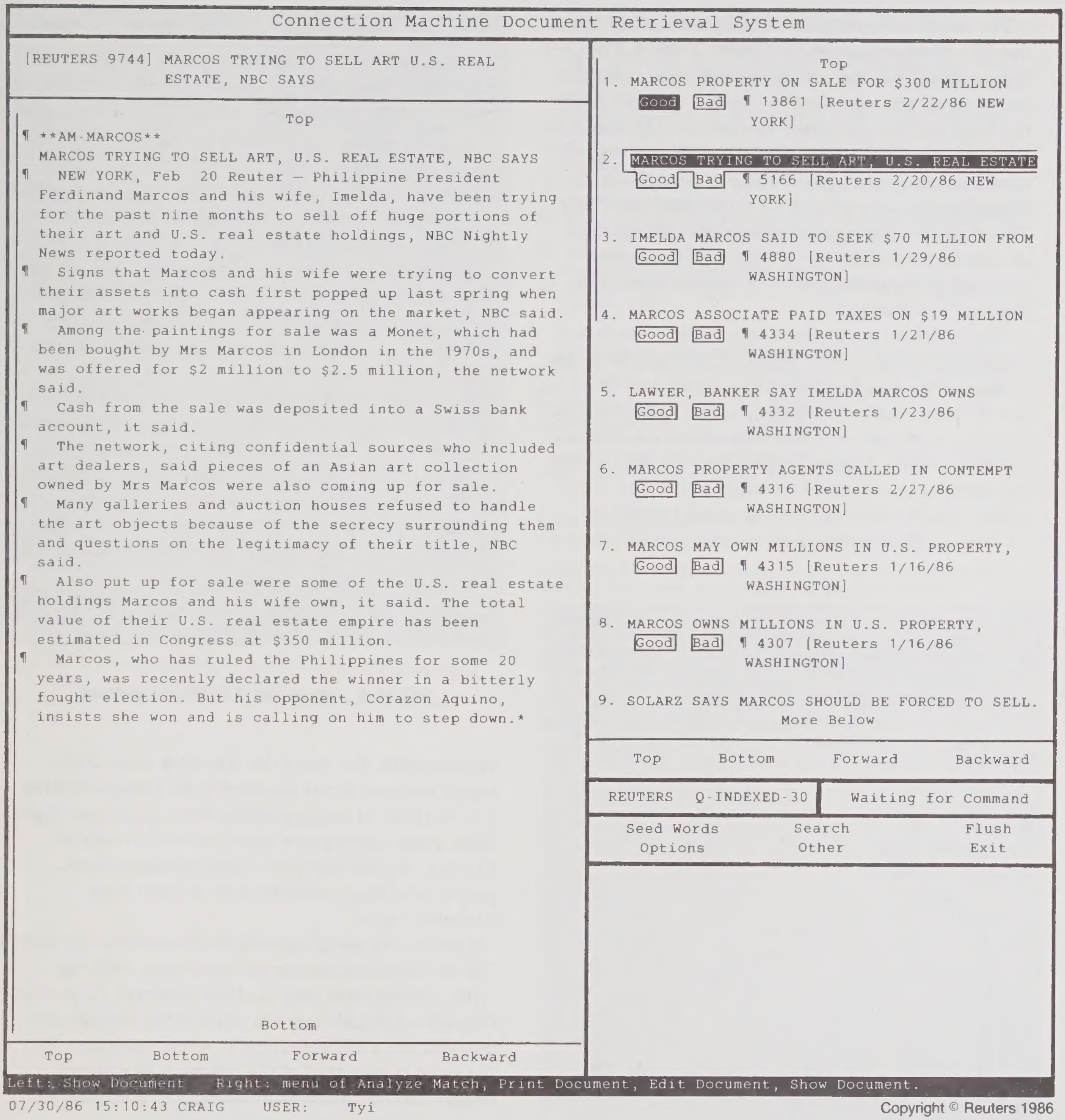
```
                     Connection Machine Document Retrieval System

[REUTERS 8594] MARCOS PROPERTY ON SALE FOR $300 MILLION,   |                    Top
               NEW YORK TIMES SAYS                         | 1. NEWS PICTURES ADVISORY
                                                           |    Good  Bad  ¶ 1125 [Reuters 1/03/86]
                       Top                                 |
¶ **PM-PRESIDENT-PROPERTY**                                | 2. AQUINO DENOUNCES MARCOS IN PRESIDENTS
  MARCOS PROPERTY ON SALE FOR $300 MILLION, NEW YORK TIMES |    Good  Bad  ¶ 1125 [Reuters 1/02/86 BAGUIO,
  SAYS                                                     |               Philippines]
¶   NEW YORK, Feb 22, Reuter - Four Manhattan buildings    |
  believed owned by Philippine President Ferdinand Marcos  | 3. MARCOS TRIES TO LAUGH OFF U.S. ARMY CHARGES
  and his wife Imelda have been on the market since last   |    Good  Bad  ¶ 1125 [Reuters 1/24/86 MANILA]
  June for up to $300 million, the New York Times reported |
  today.                                                   | 4. PHILIPPINES ASSAILS REPORT ON FAKE MARCOS
¶   All are owned by corporations situated in countries    |    Good  Bad  ¶ 1125 [Reuters 1/23/86 MANILA]
  outside the United States which keep secret the identity |
  of the owners, although Congressional testimony has      | 5. PHILIPPINES ASSAILS REPORT ON FAKE MARCOS
  indicated they are controlled by the Marcoses, the       |    Good  Bad  ¶ 1125 [Reuters 1/23/86 MANILA]
  newspaper said.                                          |
¶   It said documents from the Security Pacific National   | 6. AQUINO DEATH THREAT REPORTED AS RAIN
  Bank indicated the owners wanted to sell the properties  |    Good  Bad  ¶ 1125 [Reuters 1/21/86 MARAWI,
  because of financial difficulties, management disputes,  |               Philippines]
  and sensitivity to political criticism in Philippines    |
  about holdings in the United States."                    | 7. MARCOS PROPERTY ON SALE FOR $300 MILLION
¶   Although the documents do not reveal the potential     |    Good  Bad  ¶ 1125 [Reuters 2/22/86 NEW
  profit from the sale of the buildings, testimony before  |               YORK]
  the House Foreign Affairs Subcommittee on Asian and      |
  Pacific Affairs showed that Mrs. Marcos told one         | 8. SOLARZ TO SUBPOENA MARCOS DOCUMENTS HELD
  building manager in 1984 she hoped to make $70 million   |    Good  Bad  ¶ 1125 [Reuters 3/14/86
  by 1987, the report said.                                |               WASHINGTON]
¶   The Marcoses' wealth was a significant issue in the    |
  Feb. 7 presidential elections, although the couple has   | 9. SOLARZ TO SUBPOENA MARCOS DOCUMENTS HELD
  denied owning property in the United States. Some        |                 More Below
  congressmen have said they want to restrict further aid  |
  to the Philippines until the Marcoses repatriate their   |   Top      Bottom      Forward      Backward
  American investments.                                    |
¶   According to the report, discussions on the sale of    | REUTERS  Q-INDEXED-30  | Reading Words
  the properties began about the time that articles began  |
  to appear about their holdings last June.                | Seed Words        Search         Flush
¶   NBC reported Thursday that the Marcoses have been      |  Options          Other          Exit
  trying for the past nine months to sell off huge         |
  portions of their art and U.S. real-estate holdings.     | Seed Words> marcos hidden wealth
¶   Signs that the couple were trying to convert their     |
  assets into cash first popped up last spring when major  |
  art works began appearing on the market, NBC said.       |
¶   Among the paintings for sale was a Monet, which had    |
  been bought by Mrs. Marcos in London in the 1970s,       |
  offered for between $2 million and $2.5 million, the     |
  network said.                                            |
¶   Cash from the sale was deposited into a Swiss bank     |
  account, it said.*                                       |
                                                           |
                    More Below                             |
                                                           |
  Top        Bottom           Forward          Backward    |
L: Weighted M: Boolean And P: Boolean Or
07/30/86 15:10:18 CRAIG    USER:    Tyi
```

Copyright © Reuters 1986

FIGURE 1. Seed Word Phase of a Search

12), which were presumed to be the entire set of relevant documents in the database. Next, for comparative purposes, *conjunctive Boolean queries* of two terms were executed: For the first task, the query was "Jordan and Syria;" for the second, it was "Marcos and Wealth." The results of these queries were tabulated according to recall and precision values. Third, a *simple query* was executed for the two respective tests—"Jordan Syria summit" and "Marcos Hidden Wealth"—and recall and precision rates were tabulated for different subsets of the query result. In the first test, three tabulations were made for the best 10, 20, and 30 documents retrieved; in the second, only one tabulation was made


Connection Machine Document Retrieval System
Top
Bottom
Forward
Backward
Top
Bottom
Forward
Backward
REUTERS Q-INDEXED-30
Waiting for Command
Seed Words
Search
Flush
Options
Other
Exit
Left: Show Document Right: menu of Analyze Match, Print Document, Edit Document, Show Document.

[REUTERS 9744] MARCOS TRYING TO SELL ART U.S. REAL ESTATE, NBC SAYS

¶ **AM-MARCOS**
MARCOS TRYING TO SELL ART, U.S. REAL ESTATE, NBC SAYS

¶ NEW YORK, Feb 20 Reuter – Philippine President Ferdinand Marcos and his wife, Imelda, have been trying for the past nine months to sell off huge portions of their art and U.S. real estate holdings, NBC Nightly News reported today.

¶ Signs that Marcos and his wife were trying to convert their assets into cash first popped up last spring when major art works began appearing on the market, NBC said.

¶ Among the paintings for sale was a Monet, which had been bought by Mrs Marcos in London in the 1970s, and was offered for $2 million to $2.5 million, the network said.

¶ Cash from the sale was deposited into a Swiss bank account, it said.

¶ The network, citing confidential sources who included art dealers, said pieces of an Asian art collection owned by Mrs Marcos were also coming up for sale.

¶ Many galleries and auction houses refused to handle the art objects because of the secrecy surrounding them and questions on the legitimacy of their title, NBC said.

¶ Also put up for sale were some of the U.S. real estate holdings Marcos and his wife own, it said. The total value of their U.S. real estate empire has been estimated in Congress at $350 million.

¶ Marcos, who has ruled the Philippines for some 20 years, was recently declared the winner in a bitterly fought election. But his opponent, Corazon Aquino, insists she won and is calling on him to step down.*

1. MARCOS PROPERTY ON SALE FOR $300 MILLION
   Good Bad ¶ 13861 [Reuters 2/22/86 NEW YORK]
2. MARCOS TRYING TO SELL ART, U.S. REAL ESTATE
   Good Bad ¶ 5166 [Reuters 2/20/86 NEW YORK]
3. IMELDA MARCOS SAID TO SEEK $70 MILLION FROM
   Good Bad ¶ 4880 [Reuters 1/29/86 WASHINGTON]
4. MARCOS ASSOCIATE PAID TAXES ON $19 MILLION
   Good Bad ¶ 4334 [Reuters 1/21/86 WASHINGTON]
5. LAWYER, BANKER SAY IMELDA MARCOS OWNS
   Good Bad ¶ 4332 [Reuters 1/23/86 WASHINGTON]
6. MARCOS PROPERTY AGENTS CALLED IN CONTEMPT
   Good Bad ¶ 4316 [Reuters 2/27/86 WASHINGTON]
7. MARCOS MAY OWN MILLIONS IN U.S. PROPERTY,
   Good Bad ¶ 4315 [Reuters 1/16/86 WASHINGTON]
8. MARCOS OWNS MILLIONS IN U.S. PROPERTY,
   Good Bad ¶ 4307 [Reuters 1/16/86 WASHINGTON]
9. SOLARZ SAYS MARCOS SHOULD BE FORCED TO SELL.

More Below

07/30/86 15:10:43 CRAIG USER: Tyi

Copyright © Reuters 1986

FIGURE 2. Seed Document Phase of a Search

as only 9 documents in total were retrieved. The fourth phase of the experiment applied *seed document search* to the set of documents produced by the simple query used in phase three. A small number of the retrieved documents were marked Good (in experiment 1 (Syria), a single document; in experiment 2 (Marcos), three documents), and a search was triggered. Again, the results were tabulated for recall and precision, taking several subsets of the query result. In the fifth phase of experiment 1, each of the top 10 documents retrieved in the previous phase (i.e., by the first seed document search) was marked either Good or Bad; the test was repeated and the results tabulated.

The results of testing the techniques against the two test cases are shown in Figures 3 and 4. The figures show that the simple queries were comparable in recall and precision to Boolean queries, while the large queries generated by relevance feedback in the seed document phase of the search were significantly better in both recall and precision, than either the simple query alone or the Boolean query. This confirms previous results on the effectiveness of relevance feedback for purposes of document retrieval [10] and the problems encountered with Boolean searches [1].

The interface to our document-retrieval system is extremely simple to use. There is no query language to learn: The user need only know how to enter a list of words, browse through articles, mark them as `Good` or `Bad`, and initiate a new search—a 5-minute training session at most. In addition, the search does not involve an extensive guessing game. The initial phase, which involves locating a small initial set of relevant documents, does require a small amount of guesswork, but since the aim here is not high recall, the user is not required to be comprehensive (i.e., to think of every possible word an author might have used). Subsequent iterations do not require this sort of guessing, either, and reactions from people who have used the system have been extremely good.

	Query	Recall	Precision
A	Disjunction, 3 terms	1.00	0.05
B	Conjunction, 2 terms	0.71	0.27
C1	Simple, 3 terms, first 10	0.36	0.50
C2	Simple, 3 terms, first 20	0.57	0.40
C3	Simple, 3 terms, first 30	0.71	0.33
D1	Mark 1, first 10	0.57	0.80
D2	Mark 1, first 20	0.86	0.60
D3	Mark 1, first 30	0.93	0.43
E1	Mark 10, first 10	0.71	1.00
D2	Mark 10, first 20	0.86	0.60
E3	Mark 10, first 30	1.00	0.47

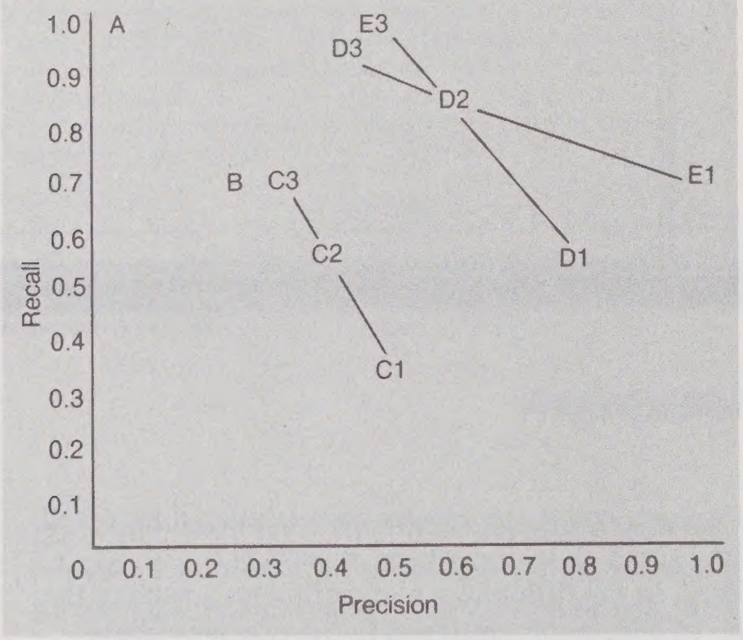


FIGURE 3. First Run of Retrieval Experiment

	Query	Recall	Precision
A	Single term	1.00	0.06
B	Conjunction, 2 terms	0.25	0.30
C	Conjunction, 2 other terms	0.17	0.33
D	Simple, 2 terms	0.25	0.30
E1	Mark 3, first 10	0.67	0.80
E2	Mark 3, first 20	0.92	0.55
E3	Mark 3, first 30	1.00	0.40

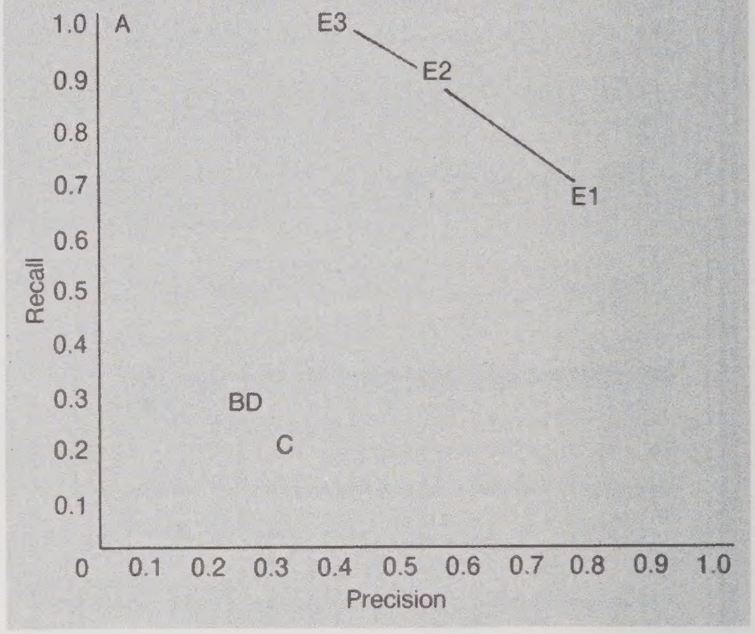


FIGURE 4. Second Run of Retrieval Experiment

Finally, the search system itself is extremely fast: For the sample database we have been working with, the response time is about a second. In general, only 60 ms of CM time are required to find the 20 best hits for a 200-term query, which indicates that the basic throughput of the machine as a search engine should be adequate to support a significant number of users on a large database.

CONCLUSION

Using the CM, it is possible to perform a fast parallel search of a database. For a database of 18 Mbytes (31,994 documents) on a 16,384 element machine, the measured time to execute a query varies from 0.004 s for a Boolean query with 25 terms to 0.295 s for a Boolean query with 20,000 terms (compute time only). For a 15-Gbyte database, the estimated time to execute a query varies between 2 min. for a Boolean query with 25 terms to 3 min. for a Boolean query of 20,000 terms (compute plus I/O time). It is also possible to search a database using simple queries and achieve similarly high search speeds: 60 ms for a 200-term query on a 112 Mbyte database.

Using these facilities, it is possible to implement database search techniques that are not feasible given sequential machines and inverted databases. A demonstration system utilizing relevance feedback has been written and tested. It combines high precision and recall with ease of use and fast response, and represents an advance over existing free-text database search technology.

Acknowledgments. Many thanks to Jim Bailey and Dave Waltz for guidance in this project, to Steven Smith for work on the user interface, and to the members of the Connection Machine System Group for providing the hardware. Special thanks to Tom Kraay for suggesting the project and to Reuters for making their wire service available to us.

REFERENCES

1. Blair, D.C., and Maron, M.E. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Commun. ACM 28*, 3 (Mar. 1985), 289–299.
2. Dodds, D.J. Reducing dictionary size by using a hashing technique. *Commun. ACM 25*, 6 (June 1982), 368–370.
3. Hillis, D. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.
4. James, G. *Document Databases*. Van Nostrand Reinhold, New York, 1985.
5. Kraay, T. Unpublished manuscript. MRJ, Fairfax, Va., Apr. 1986.
6. Mott, P.M., Waltz, D.L., Resnikoff, H.L., and Robertson, G.G. Automatic indexing of text. Tech. Rep. 86-1, Thinking Machines Corp., Cambridge, Mass., Jan. 1986.
7. Nix, R. Experience with a space efficient way to store a dictionary. *Commun. ACM 24*, 5 (May 1981), 297–298.
8. Peterson, J.L. Computer programs for detecting and correcting spelling errors. *Commun. ACM 23*, 12 (Dec. 1980), 676–687.
9. Potter, J.L. *The Massively Parallel Processor*. J.L. Potter, Ed. MIT Press, Cambridge, Mass., 1985.
10. Salton, G. *The SMART Retrieval System—Experiment in Automatic Document Processing*. Prentice-Hall, Englewood Cliffs, N.J., 1971.
11. Salton, G. Another look at automatic text-retrieval systems. *Commun. ACM 29*, 7 (July 1986), 648–656.
12. van Rijsbergen, C.J. *Information Retrieval*. 2nd ed. Butterworths, London, 1979.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*single-instruction-stream, multiple-data-stream processors (SIMD)*; H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Design, Performance

Additional Key Words and Phrases: Connection Machine, free-text databases, parallel computing, relevance feedback

Received 4/86; accepted 6/86; revised 8/86

Authors' Present Address: Craig Stanfill and Brewster Kahle, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Business Day

The New York Times

For Shakespeare, Just Log On

Large PC Libraries Are Being Developed

By JOHN MARKOFF

The development of a nationwide data network will allow personal computer users to tap sources as large as the Library of Congress or receive their own personalized electronic newspapers.

Several innovations, taken together, have already demonstrated that searching vast computer data bases can be easier than consulting a card catalogue, and not nearly as difficult or expensive as computer searches are today. Computer users might read some Dickens more readily than they could check out David Copperfield from the local library.

Those in the industry say that users with little computer skills will soon be able to search through several terabytes of information, or several trillion characters of text, in seconds. The Library of Congress, with 80 million items, contains an estimated 25 terabytes of information.

Already, an experimental computer library has linked 150 universities to 40 sources of information, ranging from National Institutes of Health data to corporate documents and Shakespeare's plays. New software allows users to browse or zero in on particular information.

As methods of retrieving information are standardized and perfected, industry executives and computer scientists say, thousands of new services, ranging from electronic newspapers to the computer equivalent of free public libraries, will blossom. "Everyone is realizing how important it is to get into the mass market for information," said Thomas Koulopoulos, president of Delphi Consulting Group, a Boston market research firm.

Such ready access to huge amounts of computerized information has been the dream of many in the industry. But a lack of computing power, effective software and high-speed digital networks has stalled progess until recently.

If many of the technical problems are being solved, major business and political disputes remain. The researchers acknowledge that they must resolve several questions of privacy and pricing before they can put the new methods to commercial use.

Many sources of information, like government documents, might be available free, but other services, including electronic newspapers, will be available only to those who pay. The industry has yet to settle on ways to protect and charge for intellectual property in a computer network where information can be copied instantly. But to encourage progress, the Thinking Machines Corporation, a Cambridge, Mass., supercomputer manufacturer, has made its software available at no charge.

Some industry enthusiasts say the new technology will transform the

Continued on Page D5

Mike Theiler for The New York Times

Brewster Kahle was the leader of the development team at the Thinking Machines Corporation for a nationwide computerized library system. His team's software links a CM2A Connection Machine, left, with a personal computer or work station like the Apple Macintosh II at right. Using high-speed data highways, the two machines can function together although they may be thousands of miles apart.

For Shakespeare, Just Log On

Continued From First Business Page

way computerized information is sold. Mitchell Kapor, the founder of the Lotus Development Corporation, predicts the growth of a new industry as significant as the personal computer business. Some companies, like Dow Jones & Company, that already provide computerized information over telephone lines have taken part in developing the new computer library.

The Search Is Simplified

In 1989, Thinking Machines enlisted the support of Dow Jones, Apple Computer Inc. and the KPMG Peat Marwick accounting and consulting firm to design the computer library, called Wide Area Information Servers, or WAIS (pronounced ways). The system permits computer users to quickly search through a huge volume of information even if it is stored at several distant locations.

The system lets users conduct searches by typing common English phrases instead of more complicated computer commands. While current systems like Dialog and Nexis require users to specify precisely the information they want, the new system can respond to a user's inferences. It initially presents a sample list of documents. The user chooses one or several, and then a "relevance feedback" program presents other documents most like the ones selected.

"This solves the problem of how to get to the information you need, getting not too much and not too little," said Esther Dyson, editor of Release 1.0, a computer industry newsletter.

> It will soon be possible to search through millions of items in seconds.

This is a sharp contrast to the way services operate today, Ms. Dyson said. A computer user may need to call seven or eight separate data bases depending on the kind of information needed.

The WAIS system lets users of Apple personal computers harness a network of Thinking Machines supercomputers and smaller "server" computers to search data bases stored by Dow Jones, KPMG and several corporations and universities. Users can also read electronic mail, enter their corporate electronic libraries and summon up a wide variety of documents, newspapers and magazines.

A 'Corporate Memory'

At Thinking Machines, the WAIS system serves as a "corporate memory," allowing employees to retrieve memos, documents and other internal information. Employees who may not be working together can share expertise.

"If someone did something in Los Angeles and I'm sitting in San Francisco, I may not know about the work," said Robin Palmer, a senior manager at Peat Marwick.

WAIS delivers information over the Internet, a collection of 2,600 high-speed public and private computer networks. This Government-sponsored system of data highways is rapidly being improved and turned to commercial uses.

The market for software that allows the rapid retrieval of computerized text is small but growing, according to industry analysts. In 1989, the United States had fewer than 60,000 users; by the next year, total sales were about $120 million. The Delphi Consulting Group expects the market to grow to 160,000 users and $235 million by 1992.

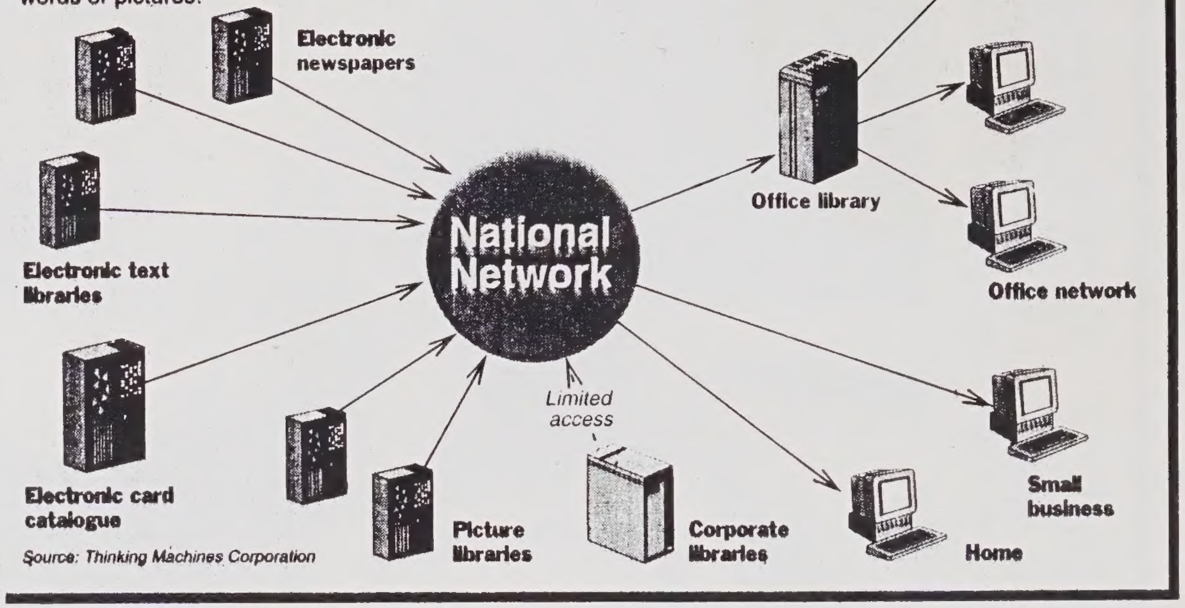
"Information retrieval technology is starting to spread from supercomputers all the way down to personal computers," said Brewster Kahle, a Thinking Machines scientist who has led the WAIS experiment.

The WAIS system is built on a procedure for retrieving information developed by librarians who initially set out to computerize their card catalogues. The procedure — known in the field as Z39.50 — now has the support of the Library of Congress, Apple, Sun Microsytems Inc., Next Inc., Dow Jones and Mead Data Central.

In the future, a special directory or "white pages" will keep an up-to-date list of all the separate sources on the network.

Spreading Information

The Wide Area Information Servers system provides a broad range of information by linking users to many independent sources The information can be in the form of sound, words or pictures.



Source: Thinking Machines Corporation

Apple has its own electronic library project, borrowing its name, Rosebud, from the movie "Citizen Kane." The three-year-old project is based on the WAIS system, but adds features including the ability for a user to develop a personalized electronic newspaper.

Rosebud uses special programs — called "reporters" — that let customers specify the kinds of information and news they want to retrieve from the WAIS system every day. Researchers at Apple's Advanced Technology Group said that in the future the necessary retrieval software might be a standard part of a computer's operating system.

They expect improvements in the Internet computer network to greatly lower the cost of information searches, promoting the introduction of many new services. The Government proposes to expand and improve Internet by financing a National Research and Education Network, or NREN, that could extend a high-speed computer links into schools and communities across the country.

"With things like NREN, everthing could change overnight," said Tim Oren, an Apple researcher.

MacWEEK
05.14.91 Volume 5 Number 19 THE NEWSWEEKLY FOR MACINTOSH MANAGE

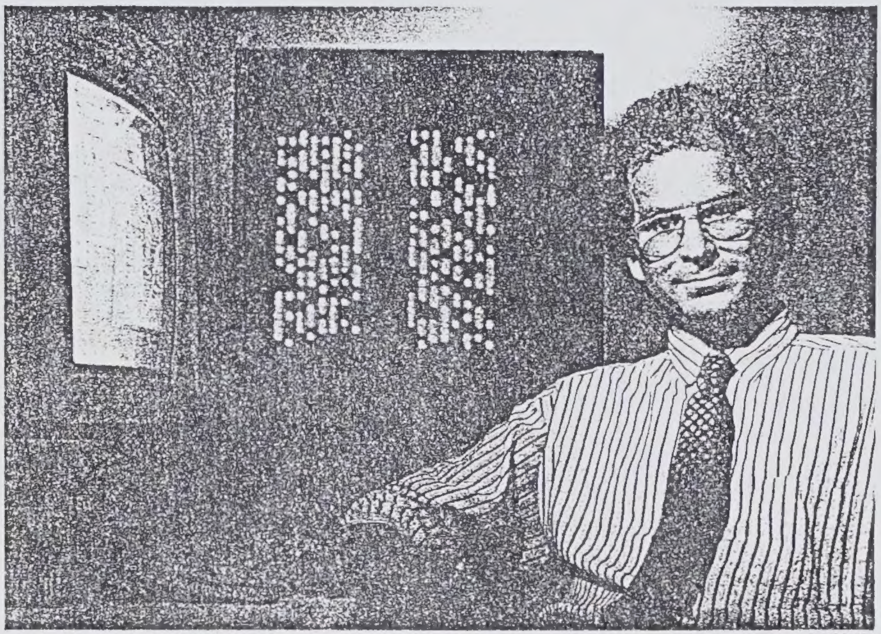
WAIS promises easy text retrieval

Prototype links Mac, Connection Machine

By Henry Norr

Cupertino, Calif. — Thinking Machines Corp., a pioneer in the development of high-powered parallel-processing supercomputers, has joined with Apple, Dow Jones & Co. and KPMG Peat Marwick to develop a new technology designed to simplify the retrieval of textual information stored in personal files, corporate records and remote databases.

Called the Wide Area Information Server (WAIS) project, the collaborative venture has been under way for almost two years. Peat Marwick recently completed a four-month experiment with the system, using WAIStation, a prototype Mac front end developed by Thinking Machines of Cambridge, Mass. Engineers from Apple's Advanced Technology Group have combined the WAIS technology with a custom interface to build a prototype personal electronic newspaper.


DRAKE SOREY

Brewster Kahle, WAIS project leader, helped develop an experimental text-retrieval system that can use a Thinking Machines supercomputer as a server and the Mac as a front end.

The WAIS project was designed in part to address problems caused by the proliferation of electronic data within large organizations.

"Corporations are starting to gag on gigabytes of word processing files, memos, reports, articles and E-mail archives," said Brewster Kahle, WAIS project leader for Thinking Machines. "Corporate memory is stored in this form, but executives have no easy way to get at it."

But the WAIS project was intended from the beginning, Kahle said, to be more than a traditional executive information system working only within corporate bounds. The objective was to lay the foundations for a scalable system that would allow users to tap a variety of data sources, including large commercial databases, through a uniform interface. Users, according to the plan, should be able to search for any available information without having to master the internal organization and query techniques of each source.

See Thinking Machines, Page 24

Peat Marwick tries 'partner-friendly' system

When officials of Thinking Machines Corp., Dow Jones & Co. and Apple first broached the concept of the Wide Area Information Server (WAIS) with KPMG Peat Marwick, representatives of the accounting giant were intrigued but cautious, according to Brewster Kahle, project leader for Thinking Machines.

They weren't interested, he said, in another complex querying application that busy tax consultants, accountants and managers would never bother to use. But they agreed to participate in the project, according to Kahle, on the promise of a system that would be genuinely "partner-friendly," with "no algebra — no ifs, ands or buts."

After a year of preliminary work, an experimental WAIS R&D project went on-line at Peat Marwick last October. About 10 users at the company's Montvale, N.J., headquarters, including "very senior partners," took part in the experiment, along with two others in Manhattan and 10 more on the West Coast, according to Robin Palmer, senior manager and WAIS project leader at KPMG Peat Marwick in San Jose, Calif. The remote users were connected by leased lines to a WAIS server running on a Connection Machine, a Thinking Machines parallel-processing system, installed in Montvale.

The Peat Marwick experiment relied on WAIStation, a Mac-based client software program developed by Thinking Machines, as a front end. To prepare a query, users need only enter the subject they are interested in, in English — "IBM and Motorola," for instance, or "recent developments in personal computers" — in a text field labeled "Look for documents about." They then drag icons representing possible sources, local or remote, into another field.

When the query is run, the Macintosh-based front end encodes the search string according to the WAIS protocol and passes it to the specified servers. Each server translates the query into its own language, locates matching articles and returns the results to the front end.

The WAIStation application then displays headlines for each article; the citations are ranked according to probable relevance, based on algorithms that consider the position, frequency and proximity of desired terms within the text.

By double-clicking on the headline, users can get the full text of any of the articles. And if the user drags the most useful titles into a bin labeled "Similar to" and reruns the search, the system will track down additional articles that share a large number of words with those selected.

Peat Marwick completed its WAIStation testing in February. In part because the cost of maintaining a real-time wide-area link among its many offices would be "substantial," according to Palmer, the company has not made a commitment to the system and is still considering a variety of alternatives. But, he said, "we are still extremely interested in the WAIS concepts. It's a most promising technology." — *By Henry Norr*

Thinking Machines

From Page 22

The WAIS system has three components:

➤ **Server software.** Any information source capable of locating and presenting text in response to a request in WAIS format can function as a server; the source can be on the user's own machine, on a LAN or at a remote site connected by modem. The WAIS client software can keep track of multiple servers, search any or all in response to a single request and consolidate the results.

Thinking Machines now includes the WAIS text-indexing and retrieval software free with its Connection Machines, a line of massively parallel systems that range in price from $100,000 to $5 million, according to Kahle. In addition, the companies participating in the project developed a sample server that runs on standard Unix systems. But any text-retrieval program on any platform, including the Mac, could be adapted to function as a WAIS server.

➤ **Protocol.** To foster the development of WAIS-compatible data sources, the four companies created an open protocol for transmitting queries and responses. It is based on an existing standard, the National Information Standards Organization's Z39.50 protocol, but is enhanced in several ways, such as by the addition of support for audio and video information.

➤ **Clients.** WAIS was designed to support a variety of interfaces running on various platforms and tailored to different niches.

The system does not rely on a specialized query language; the front end simply passes English-language search strings entered by the user to the server.

In addition to the prototype WAIStation interface and Apple's experimental personal newspaper, front ends already are available for the X Window System and GNU emacs, an extensible text editor that runs under a freely distributed Unix-like operating system developed at the Massachusetts Institute of Technology in Cambridge.

To promote the WAIS concept, Thinking Machines is making source code for the system available over the Internet or by mail. The code comes free of charge but without support. Using the software, programmers at MIT and elsewhere already have created more than 20 WAIS servers, including a poetry server, a weather server and a catalog of government programs. Thinking Machines will maintain a publicly accessible directory of servers, which will include descriptions of all known servers and special files that allow WAIS front ends to plug into them.

COMPUTER SOCIETY
PRESS REPRINT

OPTICAL INTERCONNECTS FOR INTERPROCESSOR COMMUNICATIONS IN THE CONNECTION MACHINE

Brewster O. Kahle
Edward C. Parish
Thomas A. Lane
Jerry A. Quam

Reprinted from PROCEEDINGS OF THE 1989 INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, Cambridge, MA, October 2-4, 1989

The Computer Society of the IEEE
1730 Massachusetts Avenue NW
Washington, DC 20036-1903

Washington • Los Alamitos • Brussels

IEEE
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE
COMPUTER
SOCIETY
PRESS

Optical Interconnects for Interprocessor Communications in the Connection Machine

Brewster O. Kahle and Edward C. Parish
Thinking Machines Corporation

Thomas A. Lane and Jerry A. Quam
Honeywell Sensors and Signal Processing Laboratory

Optical fibers are demonstrated effective in interconnecting processors in a massively parallel computer in circumstances where low wire density is essential. The demonstration is the execution of a Fast Fourier Transform (FFT) on the 16K processor Connection Machine® computer[1] using optical interconnect for interprocessor communications. The Connection Machine computer has 65,536 processors interconnected through a hypercube network. In commercial systems, current costs make short distance optical fibers only attractive when connector density or wire density are critical.

The prototype utilized a 400 Mbit/sec optical interconnect consisting of two fibers to provide the interprocessor communication between two 8k processor subcubes. A 1100 Mbit/sec link will be completed by the end of 1989. In this case, optical interconnects provided over two orders of magnitude improvement in "wire" density.

This paper describes the architecture, design, and physical characteristics of a prototype optical interconnect for the Connection Machine. It also explores the future use of optical interconnects in computer systems.

Interconnect Technology

Although not considered as glamorous as processor or memory design, the interconnect system is absolutely fundamental to a computer's design. In serial computers, the interconnect system is usually a backplane for inter-board connections and flat copper cables which connect to peripherals. Optical interconnect is sometimes used for local area networks (LANs). The relatively inexpensive nature of a backplane plus the lack of a standardized multi-drop optical solution make current fiber optics unattractive as substitutes for the copper competitor. In parallel machines many interprocessor or processor-memory connections are point-to-point and require high bandwidth which make optical interconnect a possible alternative to copper wires. This section will break down some of the general uses of off-board interconnect in current computer systems and compare the relative merits of flat cable, coax cable, and fiber optics for two meter circuits.

Serial machines, such as the VAX 8700 or the Sun-4/280, have separate connections from the processor to the memory subsystem and to I/O ports. These connections are under 1 meter and require multiple board connections. Backplanes are appropriate for this type of connection because they provide low latency, wide potential channels, low cost, and multi-drop capability.

Connecting serial machines together over a local area network (LAN) requires long distances as well as small "wire" dimensions. Coax cable and optical fiber have both been effectively used in this application. The speed of current LAN's of 10-100Mbits/sec is easily reached in either technology. The interface cost for coax or fiber is not significant because only one interface is needed per computer system.

Parallel computers, such as the Connection Machine or the BBN Butterfly[BBN], use many point-to-point connections for interprocessor communication or to connect processors to the memory system. This communication speed is crucial to the overall system performance. Given the large number of these connections in parallel systems (192 flat cables in the Connection Machine), cost, connection size, power consumption, and wire size become critical. Currently, copper wires are the most cost effective and reliable interconnect in parallel machines. However, as machines grow more powerful, wire density becomes critical making coax and fiber possible alternatives because of their smaller wire size.

If one wanted to construct a unidirectional 1 gigabit, 2 meter connection for CMOS parts on different PC boards, there are a few viable alternatives: flat cable of twisted pair, coax running at a gigabit/sec, coax running at 200Mbits/sec, and fiber optics. Each has a different set of characteristics and costs. The size and cost estimates in this section are approximate for a conservative design (see Table 1).

	Flat Cable	Coax	Slow Coax	Fiber
Number of wires	40	1	8	1
Cost ($\$/foot$)	1.50	1	8	1
interface cost ($)	30	800	40	2000
board area (in^2)	6	6	18	3
power (*watts*)	3	2	16	2

Table 1: Gigabit Short Distance Interconnect Options

---

[1] Connection Machine is a registered trademark of Thinking Machines Corporation.

CH2794-6/89/0000/0058$01.00 © 1989 IEEE

A flat cable of twisted pair conductors can easily transmit about 50Mbits/sec per pair; therefore, 40 wires (20 pairs) are needed to transfer 1 gigabit. The wire pitch on those cables is .050 inches. The interface chips are inexpensive standard parts which consist of a set of latches and transceiver with series termination. Three pins are needed per conductor (2 to get to the backplane, and one header of the cable) and the cost can be estimated at $.25 per installed pin.

A gigabit coax cable requires a GaAs parallel-load shift register chip. The board area and power consumption of such an interface is relatively small. The GaAs interface chips currently cost approximately $400 each. This price could fall dramatically changing the economics of this solution.

Several coax cables could be used at a slower rate so that less expensive ECL could be used; these increase the board area and power requirements, but lower the parts costs. The data could be latched into separate shift registers for each coax cable.

Fiber optics requires GaAs chips and lasers to exceed 1 gigabit/sec. The electronics can be assembled in one hybrid package. Some advantages of optical interconnect over to flat cable are freedom from impedance matching/termination, lower crosstalk, smaller cables, immunity to EMI, and dielectric isolation which eliminates ground loop problems. These components, however, are quite expensive.

Costs aside, fiber optics were incorporated into the Connection Machine to explore the implications of their use in a CPU. The next sections will briefly describe the Connection Machine, the optical link, the demonstration, and conclude with the possible future of optical interconnect in computer systems.

Connection Machine Architecture

The Connection Machine system is a data parallel computer that is designed to run data intensive applications. This system includes a conventional serial machine as a front end, many tens of thousands of data processors, and a selection of I/O devices and processors. A program executing on the front end commands all the components of the Connection Machine system including the data processors, disk system, and I/O devices. This section will describe the hardware architecture and implementation of the Connection Machine Model CM-2.

Two distinct types of parallelism can be found in today's parallel computers. **Control parallel** computers achieve increased performance by taking advantage of parallelism found in the control structure of programs. The Cray X-MP, BBN Butterfly, and CalTech Cosmic Cube [Seitz85] are examples of control parallel computers. In these machines, each processor executes a portion of the program.

Data parallel computers achieve increased performance by taking advantage of parallelism found in the data of a problem[TMC86a]. The Connection Machine system, Massively Parallel Processor[MPP], and DAP[DAP], are examples of data parallel computers. Data parallel computers consist of a single instruction engine (front end), and thousands of data processors, each having local memory, connected to a communications network over which they may exchange information with other processors. Programming a data parallel computer is more akin to programming serial machines than to programming control parallel machines. Data parallel programs have only one control sequence, and the program executes on one processor, the front end. Thus, the program is running in a familiar environment with familiar tools. The data resides in Connection Machine memory and is manipulated by the data processors, which can access the memory of other processors by using a high speed intercommunication network. Similarly the front end can access the CM memory easily and efficiently. A major difference between data parallel and serial code is that iteration over the data objects is unnecessary, as all data objects can be operated on at once.

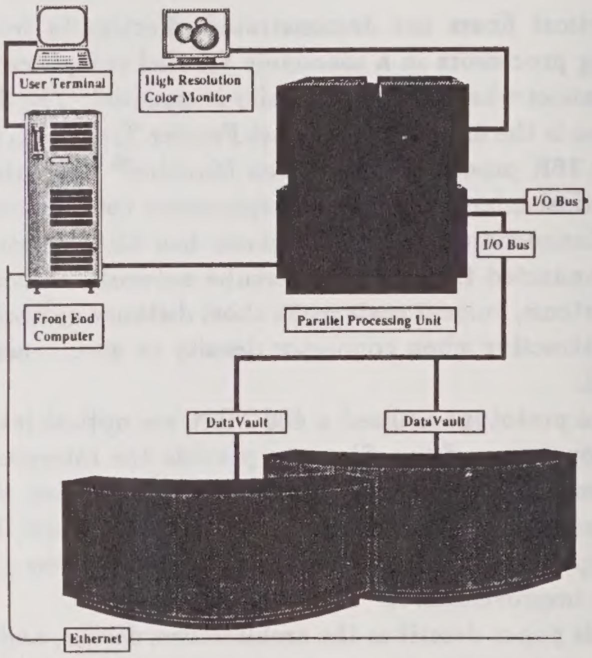


Figure 1: Connection Machine System

The Connection Machine system can consist of several front end processors, a dividable block of data processors, DataVault® disk units, high speed graphic display systems, and various I/O computers (see Figure 1).

The data processors store and process the data segments of an application. Each data processor operates on a different piece of data using the same instruction from the controlling front end processor. The data processors are very simple; a typical configuration consists of 16k - 64k processors containing 128 to 512 megabytes of memory[TMC88a].

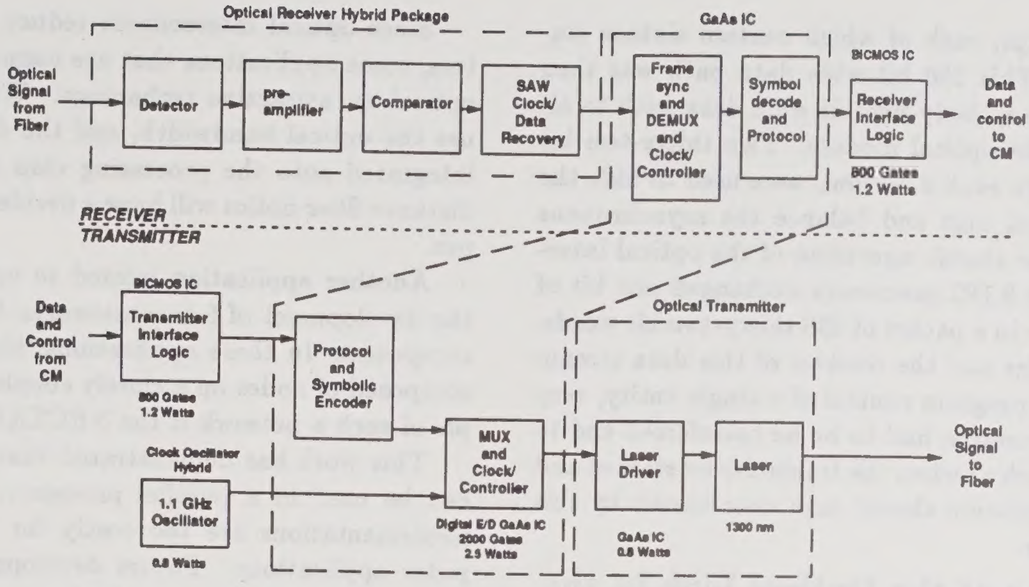

Optical Receiver Hybrid Package
GaAs IC
BICMOS IC
Optical Signal from Fiber
Detector
pre-amplifier
Comparator
SAW Clock/ Data Recovery
Frame sync and DEMUX and Clock/ Controller
Symbol decode and Protocol
Receiver Interface Logic
800 Gates 1.2 Watts
Data and control to CM
RECEIVER
TRANSMITTER
BICMOS IC
Data and Control from CM
Transmitter Interface Logic
800 Gates 1.2 Watts
Protocol and Symbolic Encoder
Clock Oscillator Hybrid
1.1 GHz Oscillator
0.8 Watts
MUX and Clock/ Controller
Digital E/D GaAs IC 2000 Gates 2.5 Watts
Optical Transmitter
Laser Driver
Laser
GaAs IC 0.8 Watts
1300 nm
Optical Signal to Fiber

Figure 2: The Gigabit Optical Link Implementation

A 64k processor machine performs a 4k by 4k matrix multiply at about 2500MFlops[TMC87]. Each processor can operate on the data within its local memory and efficiently access the memory of other processors. It is in the data processor section that the optical interconnect of this project was inserted.

The Connection Machine Model 2 is incorporates conservative chip and packaging technology to make it cost effective and reliable. The hardware is consists of 50,000 chips, 200 cables, 140 boards of 7 different types, and 25 power supplies. Almost all of the cables are used for interprocessor communication. The optical interconnect is designed to replace or augment these cables.

Interprocessor Optical Link

Optical communication has been successfully used in both telecommunications and local area network applications, and is now being explored for uses within computing systems. Possible uses include clock distribution, and interprocessor communication. The requirements for optical interconnects within these systems are different than for those used in telecommunications. Computing systems require short, high speed interconnections with small, board-mountable transmit/receive modules that introduce little additional delay, provide extremely low bit-error-rates, and dissipate little power. Optical interconnect must be compatible with standard digital packaging, power supply voltages, and environmental conditions. This section will describe how the fiber is used in the Connection Machine system.

Two versions of the CM optical interconnects were designed. The first version was built and tested at 400 Mbps with a breadboard implemented on an eight-layer printed circuit board which plugged into an I/O board in the CM-2. This breadboard contained a 32-bit input/output interface to the I/O board, protocol sequencer, clock encoder/decoder, multiplexer/demultiplexer, clock recovery, optical detector and receiver amplifier, and laser transmitter. This board used TTL, ECL, and GaAs integrated circuits. The board dimensions are 16"x9" and the power dissipation was 65 watts. Even though the individual GaAs MSI ICs could reach a clock rate 2 GHz, they ran much slower once chip-to-chip interconnects on a printed circuit board were introduced. The problem stemmed from trying to run 2.0 GHz circuitry on a 20 MHz package. Also, clock skew can be tolerated in the lower speed logic, however it becomes critical to minimize at 400 MHz.

The second module version was plug-compatible with the first but runs at 1.1 Gbits/sec and 8 watts. This module contained a 1.1 Ghz clock oscillator hybrid, a 1.1 GHz optical receiver/clock recovery hybrid, a laser transmitter hybrid, a GaAs MUX/DEMUX integrated circuit, and a BiCMOS gate array for the transmitter/receiver interface to the CM system. The printed circuit board was 5" x 5", contain one standard cell GaAs IC, two BiCMOS gate arrays, one 1.1 GHz clock oscillator, and the optical transmitter and receiver. All of the high speed logic was on the standard cell GaAs IC. The lower speed logic was on the two BiCMOS gate arrays. The BiCMOS ICs had TTL and ECL I/Os. The number of power supplies for this module were three (+5, -5, -2) since the GaAs IC that we are designing uses a single -5 V supply. An optical receiver hybrid included the detector, GaAs amplifiers, and SAW-based clock recovery. This approach eliminated the stability and resonator problems associated with a PLL, however, it required frequency stability since it cannot track large variations in the signal frequency.

The fiber connected two subcubes of the Connection Machine. Each subcube had 8,192 processors, was about two feet on a side, and was contained in one backplane assembly. The optical interconnect resided on one board in the center of the backplane with 256 copper trace connections to 512 processor chips, each of which contain sixteen single bit processors. This 256 bit wide data path was then multiplexed down to a thirty-two bit wide data path to allow connection to the optical module. Two thirty-two bit wide data fifos, one in each direction, were used to hide the latency of the optical unit and balance the asynchronous data streams. In one atomic operation of the optical interconnect, each of the 8,192 processors exchanged one bit of data with each other in a packet of 256 thirty-two bit words. Since both the sender and the receiver of this data stream operated under the program control of a single entity, very little overhead information had to be be transferred end to end. Information such as when the transmission started and how long the transmission should take were known by this controlling program.

Demonstration of the Optical Link in the Connection Machine

The optical link replaced the copper wires in one of the dimensions of the hypercube of two 8,192 processor subcubes to perform a Fast Fourier Transform (FFT). The goal of the project was to perform an FFT faster on two subcubes connected by an optical fiber than the program could be done on one subcube. This was easily accomplished with the 400 Megabits/sec optical link, thereby demonstrating the extended speed and memory size of current supercomputers when optical interconnect is used.

The FFT was chosen because of its explicit use of the dimensions of the hypercube. The straight-forward implementation of the FFT mapped a point onto each processor. Each swap over the butterfly interconnection of the FFT corresponded to a hybercube wire first connecting points that are 1 away, 2 away, 4 away, etc. All swaps were done on copper wires except for the last one which was done over the optical interconnect. A larger FFT could be run on two subcubes because of the machine's increased memory capacity.

Future of Optical Interconnects in Computer Systems

Optical communications is used successfully today in telecommunications and LAN applications[OPT]. However, the unit cost of an optical interconnect must decrease drastically before this technology is practical inside computer systems. A computer contains many interconnects, and the cost of competitive techniques, like copper wires, is very low. Clearly, optical interconnects will only be used in applications in which copper wires cannot meet the performance requirements. In future computer systems, as speeds increase and size decreases, the interconnect densities and data rate requirements may render copper wires infeasible. The cost of optical interconnects will decrease as technology provides the means to implement the optics and electronics monolithically. The costs will also decrease as developments and usage in associated areas, like LANs, provides an opportunity for manufacturing in large quantities.

Since optical interconnect reduce the need for connectors, some applications that are connector limited will find optical an attractive technology. When digital chips can use the optical bandwidth, and the drivers can be reliably integrated onto the processing chip or hybrid, then short distance fiber optics will have a decided advantage over copper.

Another application related to optical interconnects is the development of future networks for large grain parallel computers. In these applications, the computer system is composed of nodes on a closely coupled network. An example of such a network is the NECTAR system[NEC].

This work has demonstrated that optical interconnects can be used in a parallel processor, however the current implementations are too costly for the commercial computer applications. Future developments should improve the speed, size, and power dissipation characteristics of optical transmitters and receivers, which will provide more applications for optical interconnects.

Acknowledgments

This work was supported by the Defense Advanced Research Project Agency monitored by the Naval Air Systems Command under contract number N00019-87-C-0207. Thank you to Rick Epstein, Soroush Shakibnia, and Sandy Currier of Thinking Machines Corporation and Dick Schulze, Anis Husain, Steve Palmquist, Mark Bendett, Mike Wyatt, and Doug Strahm from Honeywell.

References

[DAP] Flanders, P.M. et al, "Efficient High Speed Computing with the Distributed Array Processor," *High Speed Computer and Algorithm Organization*, Academic Press, 1977, pp. 113-127.

[Seitz85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, January 1985.

[TMC86a] Thinking Machines Corporation, "Introduction to Data Level Parallelism," Thinking Machines Corporation Technical Report 86.14, April 1986.

[MPP] Batcher, Kenneth E. (1980). "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, C-29 (9).

[TMC88a] Douglas, Kahle, Vasilevsky, "The Architecture of the CM-2 Data Processor," Thinking Machines Corporation Technical Report HA88-01, April 1988.

[TMC87] Thinking Machines Corporation, "Model CM-2 Technical Summary", Thinking Machines Corporation Technical Report HA87-4, April 1987.

[BBN] Bolt Beranek and Newman Inc. "Development of a Butterfly Multiprocessor Test Bed," Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[OPT] Lane, T. A. et al, "Digital system applications of optical interconnections," *Proc. SPIE O-E/Fiber Lase*, September, 1988.

[NEC] Arnould, E. A. et al. "The Design of NECTAR: A Network Backplane for Heterogeneous Multicomputers," *Proc. Third Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, April, 1988.

The Connection Machine Model CM-1 Architecture

Brewster A. Kahle
W. Daniel Hillis

Reprinted from
IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS
Vol. 19, No. 4, July/August 1989

IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS, VOL. 19, NO. 4, JULY/AUGUST 1989

The Connection Machine Model CM-1 Architecture

BREWSTER A. KAHLE AND W. DANIEL HILLIS

Abstract —**Massively parallel computer systems offer substantially improved performance over conventional supercomputers for many applications. Designed with many thousands of processing elements operating in parallel, these machines offer a high level of efficiency for computation-intensive algorithms. Machine architectures differ with regard to storage capacities and bandwidths, and these elements, along with software overhead, determine performance. The Connection Machine Model CM-1, is presented, a massively parallel computer that exploits data parallelism for its performance. The performance of the CM-1 system is examined with the use of a simple application. Special consideration is given to the effect of problem size and data word size on performance.**

I. INTRODUCTION

THE PURPOSE of this paper is to examine the performance of the Connection Machine® Model CM-1 system, a general-purpose computer capable of adapting to many sizes and types of data sets. It will provide an overview of the hardware and software systems and present a simple application from which the raw performance of the machine can be examined. This will illustrate the power of data level parallelism and indicate how CM-1 performance is enhanced by the use of virtual processors and variable word size.

The Connection Machine system achieves its speed by operating on many thousands of data objects in parallel, rather than looping over them serially. The Connection Machine processors work with a conventional serial front-end computer as a controller, which issues the instructions executed by the Connection Machine computer. This paper describes the first commercial implementation of Connection Machine architecture, the Thinking Machines Corporation Model CM-1.

Other computers have been proposed and built that have varying similarities to the CM-1 architecture. Some machines differ in their control, such as the BBN Butterfly [1] and the NYU Ultracomputer [2]. Other machines differ in their intercommunication networks, such as the Massively Parallel Processor [6], Non-Von [8], Dado [9], and ICL DAP [10]. Most of the ideas in the CM-1 system have their roots in the long history of both parallel and serial processors that have been proposed and built. The similarities and differences with these machines will not be covered in this paper.

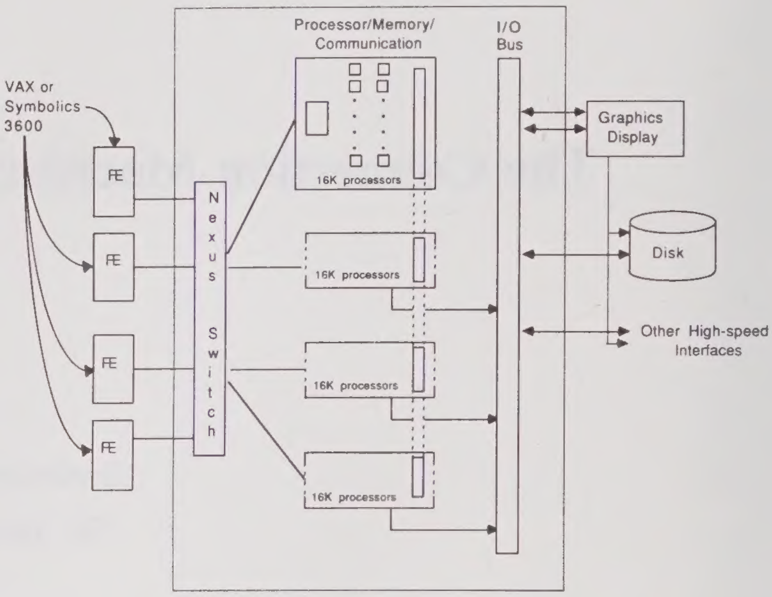


Fig. 1. Connection Machine system.

II. HARDWARE OVERVIEW

The CM-1 system consists of a processor array, from one to four front-end computers, and high-speed peripherals such as disks and image devices (see Fig. 1). The processor array contains 64 000 processing elements, each a simple serial processor with 4000 bits of memory giving 32 Mbytes of total memory for the machine. A full CM-1 system is made up of four sections of 16 000 processors each. The sections can be used separately, in pairs, or as one 64 kbyte processor unit.

Aside from its processors, each section contains interprocessor communication hardware, a sequencer, and, optionally, a set of peripherals. The interprocessor communication hardware is used for communication between any processors controlled by one host. A sequencer receives macroinstructions from the front end and broadcasts sequences of microinstructions to all the processors in its section. A typical macroinstruction would be to add two 32-bit numbers and store them in a particular location. Because of the simplicity of the processors, each macroinstruction is typically implemented by many microinstructions.

The front end computer, or host, attaches to the microcontroller through a bidirectional crossbar switch called a

---

Manuscript received April 27, 1987; revised October 31, 1987.

The authors are with Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1280.

IEEE Log Number 8427235.

®Connection Machine is a Registered service mark of Thinking Machines Corporation.

0018-9472/89/0700-0707$01.00 ©1989 IEEE

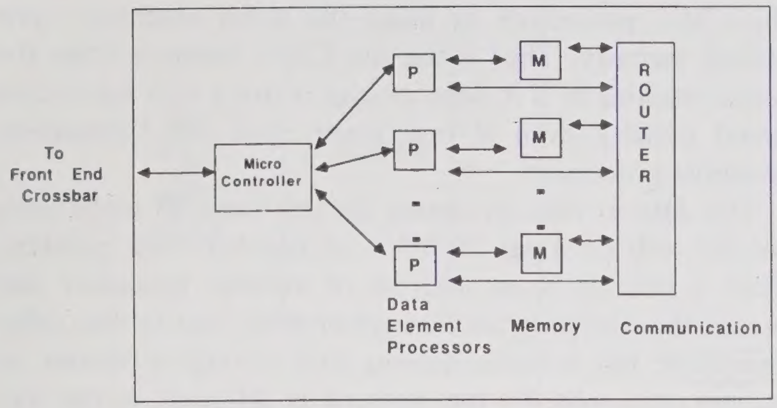


Fig. 2. Processor/memory/communication system.

Nexus, and control one, two, or four 16-kbyte processor sections. Typical front ends are the VAX® or the Symbolics 3600® Series computer.

Although the front end can be used to supply data to the CM-1 computer, in many applications the CM-1 processors can process data much faster than the front end can supply it. For this reason, the CM-1 processors are connected to a high-speed bidirectional bus consisting of up to eight I/O channels, each with a bandwidth of 320 Mbits/s. Special disk drives, frame buffers, frame grabbers, and specialized I/O devices can effectively use this bandwidth. This large I/O bandwidth makes the machine effective in very large data-intensive problems such as data bases, image analysis, and graphics.

Pointer referencing (or referencing the data in one data object from another) on the CM-1 computer requires interprocessor communication. Since a CM-1 processor needs to be able to access data from any other processor, this intercommunication system has to handle a large load at high speed. This is achieved by the use of a router which is integrated into the architecture so that every processor's memory is easily accessible to every other processor. The result is that the application programmer does not have to worry about physical processor geometry since the router handles all interprocessor communication efficiently regardless of layout (see Fig. 2).

Along with general pointer referencing, two-dimensional interprocessor communication is supported. In this type of communication, each data object can communicate with its two-dimensional neighbor (to the north, east, west, or south). This intercommunication is handled by a slightly different mechanism and is faster than using the general communication mechanism. Image processing applications often uses this ability to move data from one pixel to the next. The performance of these individual elements will be treated later in this paper.

III. SOFTWARE MODEL

From a programming point of view, the CM-1 system can be viewed as a smart memory that can be addressed and controlled by a front-end computer. The control of the program lies solely in the front end, while data are distributed between the front end and the CM-1 computer, depending on where it can be operated on most efficiently.

This section is intended as an introduction to programming the CM-1 computer. It will describe how data map onto the CM-1 processors and the types of operations that can be performed on CM-1 data. Finally, this section will present an example application and show how it can be implemented on the CM-1 system.

A. The CM-1 Computer as Smart Memory

At the heart of any large problem is the data set consisting of some combination of interconnected data objects such as numbers, characters, records, structures, and arrays. In any application these data must be selected, combined, and operated on. Data parallelism takes advantage of the parallelism inherent in large data sets.

To make effective use of data parallelism, the CM-1 computer has to be made to "look" like the data of the problem being solved. To do this, the processors in the machine are each assigned one data object, which can be any combination of immediate data and pointers. Operations can be specified to operate simultaneously on any or all data objects in the machine.

The CM-1 processors should be used whenever an operation can be performed simultaneously on many data objects. Data objects are left in the CM-1 computer during execution of the program and are operated on in parallel at the command of the front end. This differs from the serial model of reading each data object from its memory one at a time, operating on it, and then storing it again.

The front-end computer handles the flow of control, storage, and execution of the program, and all interaction with the user and/or programmer. The data set, for the most part, is stored on the CM-1 computer. The front end can fetch and store memory in individual processors efficiently to perform serial operations if this is desirable.

There are several direct benefits to maintaining program control only on the front end. First, programmers can work in an environment familiar to them. Programming languages, programming concepts, and program development support (compilers, debuggers, editors, etc.) do not change. Second, most of the existing code for an application does not have to be changed to use the CM-1 processors. A large part of most application programs pertains to the interface between the program, the user, and the operating system. Since the control of the program remains on the front end, code developed for these purposes is useful with or without the CM-1 computer, and only the code pertaining specifically to the data residing on the Connection Machine computer needs to be coded to use the CM-1 processors. Parts of the program which are especially suited for the front end, such as file manipulation, user interface, and serial data operations, can be done on the front end, while the parts of the program that run efficiently on the CM-1, namely the "inner loop" that operates on the data set, can be done there. In this way, the individual strengths of both the serial front end and Connection Machine computer can be maximized.

® VAX is a Registered service mark of Digital Equipment Corporation.
® Symbolics 3600 is a Registered service mark of Symbolics, Inc.

Most large problems have a large data set composed of interconnected structures, arrays, and records which can be mapped directly onto CM-1 processors. *Each processor's memory in the CM-1 computer holds the data from one record or array element and some small stack space for temporary results.* A processor can hold data of different types such as integers, floating point numbers, symbols, or pointers. Typically, each processor only needs 128–1024 bits for this information. Rarely is running out of memory within one processor a problem because each physical processor contains 4000 bits of memory.

Problems of different sizes will use different numbers of processors. Some problems will allocate and deallocate processors during the course of running the program. Because the CM-1's flexibility this works well. If a program needs fewer than 64 000 processors, then some sit idle, ignoring the instructions being sent by the front end. If a program needs more assigned processors than 64 000, then many more "virtual processors" are available (see Section VI).

Examples of data objects used in the CM-1 computer and pixels, transistors, data base records, atoms, and sections of physical space. Each of these are allocated one per processor. (Throughout this paper the terms "processor," "data object," and "record," are used interchangeably.)

B. Program Example

Consider the problem of manipulating a data base of employee records where each employee is a structure of employee ID, title, salary, a pointer to their boss, and a pointer to their assistant (if they have one). The example operations are "give each manager that earns over $100 000 a pay cut of $5000," give all managers' assistants a ten percent raise," and "sum all the managers' salaries" to illustrate selection and parallel operation, pointer reference, and global operations respectively.

For this sample problem, each employee's record would be assigned to a processor. On a serial machine, data are loaded from a file and put into records in the virtual memory of the serial machine. On the CM-1 system, the difference is the records are kept on the CM-1 computer.

The language *Lisp® pronounced star-lisp) is used in this paper to describe the pieces of code for the CM-1 computer. All examples can run on the CM-1 computer. In *Lisp the employee structure might look like this:

```
(*defstruct (employee))
   employee-ID;   type: number
   position;      type: number
   salary;        type: number
   boss;          type: pointer
   assistant;     type: pointer.
```

This declaration indicates that structures of this type should be created and manipulated in the CM-1 processors. If the creation and manipulation were done with the serial front end, then these operations would take about as long using the CM-1 processors as using the serial machine's own virtual memory. Thus using the CM-1 memory from the serial machine as if it were its own is not a very significant speed penalty even if one never used the Connection Machine processors.

The data in each processor (in this case, an employee's record) will be a set of fields of numbers and pointers. That a pointer is an address of another processor just means that the program can get or send data to that other processor; this is called moving data through a pointer, or pointer reference. So far, nothing is different in the way the CM-1 computer feels or performs from programming the front end. In fact most programs look very much the same whether the data are on the CM-1 computer or not (see Fig. 3).

Consider the first operation, to select all managers that earn over $100 000 and decrease their salary $5000. To execute this we must select all the data objects that satisfy the predicate "is a manager and makes over $100 000." This selection is done by instructing each processor to be "active" when the value in the employee-position slot is equal to *manager* and the value in the employee-salary slot is greater than $100 000. Notice that *manager* is a value known by the front-end and is broadcast to each processor to compare against its local data. In *Lisp the front-end code might look like this:

```
(*when (and ( = !! position (!! *manager*))
            ( > !! employee-salary 100000)))
   (*set employee-salary (-!! employee-salary (!! 5000))).
```

Executing this code on the front end would deliver an instruction stream that would look something like:

```
(let ((saved-context-bit *stack-index*))
   (CM:store-context-always saved-context-bit)
   (CM:u= constant (pvar-location position) *manager*
    8)
   (CM:and-context-with-test)
   (CM:u> constant (pvar-location employee-salary)
    100000 20)
   (CM:and-context-with-test)
   (CM:u-constant (pvar-location employee-salary) 5000
    20)
   (CM:load-context-always saved-context-bit)).
```

Notice that these instructions look like a serial computer's macroinstruction set. We have selected a set of processors and executed an operation within those processors.

To perform the "give each managers' assistant a ten-percent raise" instruction requires communication between processors. This is done by referencing data in the processor pointed to by a field in another processor. In other words, a processor would have an address (pointer) in one of its slots to another processor.

This operation is done in the following steps:

- each manager's processor gets the salary information from their assistant (in the program this is "pref!!"; or parallel reference);

® *Lisp is a Trademark of Thinking Machines Corporation.

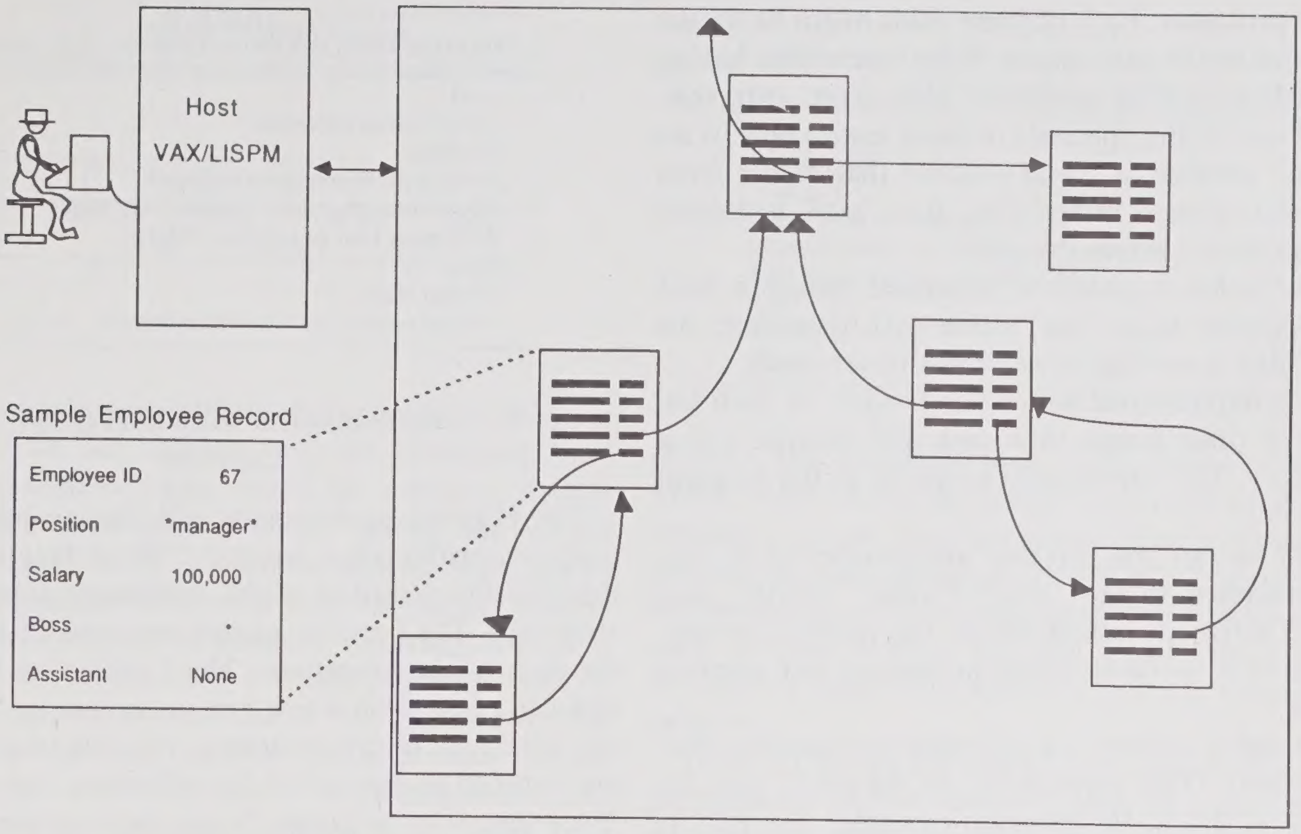


Fig. 3. Connection Machine processors: software view.

- the manager's processor saves that information in a temporary location (this is a move with the processor to the stack (the "*let"));
- the new salary is computed by multiplying the old one by 1.10 in each manager's slot (the "*!!");
- the result is sent back to the assistant's processor and put in the salary slot. This involves another move through a pointer. (This is the pset, or memory parallel set.)

This operation is done, in *Lisp, as

```
(*let (assistants-salary!! (pref!! assistant employee-
   salary))
   (*pset assistant employee-salary
      (*!! assistant-salary!! (!!0.10)))).
```

The operation, "total all managers' salary," returns a number to the front-end machine which is the total of all managers' salary slots. This is done by selecting all managers and then doing a *global sum*. Global sum is implemented using a logarithmic time tree reduction using the intercommunication wires directly. In *Lisp this looks like:

```
(*when ( = !! position (!! *manager*))    ;select managers
(*sum employee-salary))                   ;return sum.
```

This is an example of the use of *scan* that is discussed in the next section. This example shows how a specific example might map onto the Connection Machine processors and how common operations are used on such data.

IV. Basic Performance of the CM-1 System

The performance of the CM-1 system depends on the relative speeds of different common operations. Each application uses a different mix of instructions, so exact application performance depends on processor utilization and the weighted average of the instructions used. The performance rates shown in Table I describe 32-bit operations from the point of view of an active processor as a ratio to the speed of an add. Further, this section describes how these numbers scale with different arithmetic precisions and "virtual processor" ratios.

TABLE I
Relative Times of Various 32-Bit Operations

Add	1
Move within processor	0.6
Multiply	32
Move with nearest grid neighbor	4
Move through pointer (pointer reference)	25
I/O move (our or into the CM-1)	20
Scan	25
Global mix	2
Global sum	25

Performance per processor is the operation speed of an active processor or data object. A 32-bit integer addition, for instance, takes about 32 μs per processor. The system performance depends on both the speed per processor and the number of active processors. Thus the peak system performance in additions is about 2000 million instructions per second (MIPS). Other elements that influence performance are the instruction mix used by the application, the precision of the arithmetic and the total number of processors used. Some applications, such as image processing and cellular automata, run significantly faster than 2000 MIPS because they commonly use instructions on less than 32 bits. Table I uses 32 bits as a unit of measurement for other performances.

Table I describes the machine as a set of ratios with add time. Each entry is described in the following.

The 32-bit add is the unit in Table I. An add is a typical logical operation where the operands and results stay within each processor. Each of these fields might be on the stack or a slot in the data object. Other operations having similar speeds are logical operations such as OR, AND, XOR, etc. Also, if one of the operands to these instructions is an "immediate" number (a literal number that comes from the instruction stream rather than from local processor memory) the time is about the same.

The move within a processor command moves a field from one location to another within each processor. An example of this is moving an argument to the stack.

Multiply is implemented with a "shift-add" for each bit, so it takes 32 times longer than each add because it is a 32-bit multiply. This corresponds to the *!! in the program example.

Move with nearest grid neighbor are transfers of 32 bits with the processor to the "east," "west," "north," and "south." All active processors will do this transfer at once. This operation is useful in image processing and graphics applications.

Move through a pointer is a reference into another processor's memory. This corresponds to the pref!! (get the assistants' salaries) in the program example. The performance of this operation varies only slightly with the layout of the processors. Of course, if the problem is a grid problem, then the grid communications should be used instead.

I/O move is the time required to get the I/O port on the CM-1 computer. The real I/O performance will depend on the I/O device being used because most will not sustain a 320-Mbit/s/8000 processor rate. The important aspect in this performance is its balance with the other performances so that the CM-1 computer can be connected directly into data-intensive applications.

Scan is an operation that performs a form of processor intercommunication very quickly. Add–scan, for instance, results in each processor getting the sum in all processors before it. This is implemented with a parallel algorithm. Max–scan leaves the maximum up to that point:

Processor number	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	0	0	1	1	2	1	0	1
Result of add-scan	0	1	2	2	2	3	4	6	7	7	8
Result of max-scan	0	1	1	1	1	1	1	2	2	2	2.

This is a useful operation on the CM-1 computer for implementing other instructions such as global-sum but many uses have little analogy with serial software cliches.

Global-max results in the maximum value held by any active processor being passed back to the front end computer. The speed of this instruction is due to special hardware. Executing an AND of a field in all processors uses the same mechanism and runs at approximately the same speed. Similarly, OR and MINIMUM perform about the same.

Global-sum is the sum of all values held by processors in the CM-1 computer. This is implemented using *add–scan* and taking the highest processor's result.

TABLE II
RELATIVE TIMES FOR OPERATIONS ON N-BIT NUMBERS

Add	N
Move within processor	N
Multiply	N^2
Move with nearest grid neighbor	N
Move through pointer (pointer reference)	N
I/O move (out or into the CM-1)	N
Scan	N
Global max	N
Global sum	N

V. PERFORMANCES WITH DIFFERENT WORD LENGTHS

The effective performance will change based on what precision arithmetic is needed. "Word length" is used to describe the precision of the arguments to the CM-1 instructions. The previous section assumed 32-bit arithmetic for each of the operations. The Connection Machine system can operate on a few bits up to hundred of bits with the same ease or programming. Variable length arguments are handled in microcode for efficiency and ease of programming. The language *Lisp, for example, can extend the word length of numbers as needed. Sometimes a program will need only 16-bit numbers to start, but will use 32 or 64-bit numbers later (or 53 bits for that matter). It is, of course, desirable to minimize the precision required to maximize performance and minimize space utilization.

As one can see in Table II, most operations scale linearly with the word length. Thus if only 16 bit additions are needed, twice as many can be done in the time of a 32-bit addition. There is a small amount of calling overhead in each instruction call, so behavior is not quite linear for very short numbers.

Multiplication uses a shift–add algorithm so that the precision scales with the square of the number of bits; thus a 16-bit multiply would take 1/4 the time of a 32-bit multiply. The CM-1 system's flexibility extends to the precision of the arithmetic needed. Image processing applications often use very limited precision numbers, and numerical simulation often use very large precision numbers.

VI. VIRTUAL PROCESSOR CONCEPT

As presented so far, the CM-1 system has a constant (independent of problem size) performance of an add operation for problem sizes up to 64000 processors. In real applications, often 64000 processors are not enough. For example, an image processing application with a 1000× 1000 image would require one million processors, one for each pixel. To get around the arbitrary limit of the number of physical processors, the CM-1 supports virtual processors (see Fig. 4). Virtual processors are a software abstraction, implemented at the microcode level, which allows a programmer to write programs that are independent of the number of physical processors that the CM-1 hardware contains. This section will describe how virtual processors are implemented on the CM-1, and what effect they have on program performance.

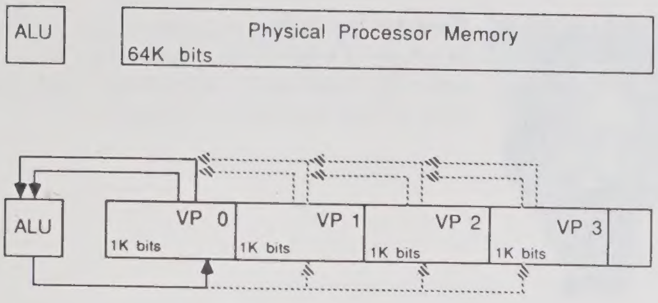


Fig. 4. Virtual processor (VP) concept.

TABLE III
RELATIVE TIME OF OPERATIONS WITH K VIRTUAL PROCESSORS ($K = 1000$)

Add	K
Move within processor	K
Multiply	K
Move with nearest grid neighbor	K
Move through pointer (pointer reference)	$K \log K$
I/O move (out or into the CM-1)	K
Scan	K
Global max	K
Global sum	K

Virtual processors are implemented using three separate mechanisms: one for storage, one for processing, and another for communications. First, the memory of each physical processor is divided evenly among the virtual processors assigned to it. The number of virtual processors per physical processor is referred to as the virtual processor ratio. For example, a problem requiring 1M (2^{20}) virtual processors would need a virtual processor ratio of 16 to operate on a 64000-machine ($2^{20}/2^{16} = 2^4 = 16$). This means that the physical address space of each virtual processor would be divided into 16 equal address spaces, one for each virtual processor assigned to it. Image processing applications commonly use only 100 bit/virtual processor so 32 processors can be assigned to each physical processor, and the CM-1 computer can be configured as a two million-processor machine.

The second mechanism necessary to support virtual processors is time-multiplexing of the physical processors among the virtual processors assigned to it. Every macroinstruction sent by the front end is run on each of the virtual processors within each physical processor. The overhead for switching context is extremely small (about the time to execute a 2-bit add) because each processor is so simple. In the 1M processor example given above, each macro instruction would be executed 16 times by each physical processor.

The third mechanism of communications allow the CM-1 processors to communicate with one another without regard to virtual processors. Grid communications are handled by the microcode by sharing the grid wires. General communications (pointer reference) is handled by the router hardware. The length of a processor address changes based on the number of virtual processors in the machine. This virtual address is used by the router hardware to deliver messages to the correct virtual processor.

Since the virtual processor abstraction is implemented at the microcode level, the virtual processing mechanism is both efficient and totally transparent to the programmer. These characteristics make the CM-1 system appropriate for problems of very different sizes. In general, it is not necessary to know how many physical processors are in a CM-1 computer.

A. Effect of Virtual Processors on Performance

Since a physical processor is shared among several virtual processors, the performance from a virtual processor's point of view goes down. If one measures the performance of the CM-1 computer as a whole, however, in terms of MIPS or MFLOPS, the performance does not go down but remains constant. In other words, given a virtual processor ratio of 16, each virtual processor is only getting 1/16 of the physical processors time, so that each macroinstruction takes 16 times as long, but each macroinstruction is doing 16 times as much work as in the case of a virtual processor ratio of 1.

Table III shows the effects of virtual processors on the performance of the various classes of macroinstructions. As expected, most operations slow down linearly with the virtual processor ratio. The only exceptions are those that involve interprocessor communication which slow by $O(K \log(K))$ for virtual processor ratios less than 16. The performance for very high ratios has not been studied. The addition of the $\log(K)$ term is due to the increase in router congestion.

Virtual processors free the programmer from the physical size of the CM-1, and also from the size of the data set that the program is to operate on. Since this is done at a very low cost in speed and programming, virtual processors have been used in most applications programmed for the CM-1 system.

VII. CONCLUSION

The model CM-1 system offers a high level of performance due to its flexible massively parallel architecture. Much of the maximum peak performance of 2000 MIPS can be used on a problem due to the balanced performances of all common instructions. The CM-1 can be used for very different types of applications because the machine can be efficiently reconfigured to have the needed number of processors and the needed arithmetic precision.

Since the CM-1 computer works closely with a serial front end computer, the programmer operates in a familiar environment. The CM-1 system combines the benefits of programming the front end computer with the performance of a massively parallel machine.

REFERENCES

[1] Bolt Beranek and Newman, Inc., "Development of a butterfly multiprocessor test bed," Rep. 5872, Quarterly Tech. Rep. 1, Mar. 1985.

[2] A. Gottlieb *et al.*, "The NYU ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, pp. 175–189, Feb. 1983.

[3] W. D. Hillis, *The Connection Machine*. Cambridge, MA: MIT Press, 1985.

[4] "Introduction to Data Level Parallelism," Thinking Machines Tech. Rep. 86.14, Apr. 1986.

[5] C. Lasser and S. Omohundro, "The essential *Lisp manual," Thinking Machines Corp., July 1986.

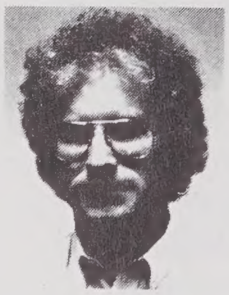
[6] D. H. Schaefer, J. R. Fischer, and K. R. Wallgren "The massively parallel processor," *Eng. Notes*, vol. 5, no. 3, pp. 313–315, May–June 1982.

[7] H. J. Siegel *et al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, Dec. 1981.

[8] D. E. Shaw, "The NON-VON Supercomputer," Dep. Comput. Sci., Columbia Univ., New York, NY, Aug. 1982.

[9] S. J. Stolfo and D. E. Shaw, "DADO: A tree-structured machine architecture for production systems," Dep. Comput. Sci., Columbia Univ., New York, NY, Mar. 1982.

[10] P. M. Flanders *et al.*, "Efficient high speed computing with the distributed array processor," in *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. New York: Academic, 1977, pp 113–127.



Brewster A. Kahle is a Scientist at Thinking Machines Corporation, Cambridge, MA. He is currently working on architectures and applications for the Connection Machine.

W. Daniel Hillis is a Founding Scientist of Thinking Machines Corporation, Cambridge, MA, currently working on machine learning and parallel computing. He is the principal architect of the Connection Machine computer.

Mr. Hillis won a 1985 ACM Distinguished Thesis Award for the book *The Connection Machine*.

T-SMC/19/4//27239

The Connection Machine Model CM-1 Architecture

Brewster A. Kahle
W. Daniel Hillis

Reprinted from
IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS
Vol. 19, No. 4, July/August 1989

The Connection Machine Model CM-1 Architecture

BREWSTER A. KAHLE AND W. DANIEL HILLIS

Abstract—**Massively parallel computer systems offer substantially improved performance over conventional supercomputers for many applications. Designed with many thousands of processing elements operating in parallel, these machines offer a high level of efficiency for computation-intensive algorithms. Machine architectures differ with regard to storage capacities and bandwidths, and these elements, along with software overhead, determine performance. The Connection Machine Model CM-1, is presented, a massively parallel computer that exploits data parallelism for its performance. The performance of the CM-1 system is examined with the use of a simple application. Special consideration is given to the effect of problem size and data word size on performance.**

I. Introduction

THE PURPOSE of this paper is to examine the performance of the Connection Machine® Model CM-1 system, a general-purpose computer capable of adapting to many sizes and types of data sets. It will provide an overview of the hardware and software systems and present a simple application from which the raw performance of the machine can be examined. This will illustrate the power of data level parallelism and indicate how CM-1 performance is enhanced by the use of virtual processors and variable word size.

The Connection Machine system achieves its speed by operating on many thousands of data objects in parallel, rather than looping over them serially. The Connection Machine processors work with a conventional serial front-end computer as a controller, which issues the instructions executed by the Connection Machine computer. This paper describes the first commercial implementation of Connection Machine architecture, the Thinking Machines Corporation Model CM-1.

Other computers have been proposed and built that have varying similarities to the CM-1 architecture. Some machines differ in their control, such as the BBN Butterfly [1] and the NYU Ultracomputer [2]. Other machines differ in their intercommunication networks, such as the Massively Parallel Processor [6], Non-Von [8], Dado [9], and ICL DAP [10]. Most of the ideas in the CM-1 system have their roots in the long history of both parallel and serial processors that have been proposed and built. The similarities and differences with these machines will not be covered in this paper.

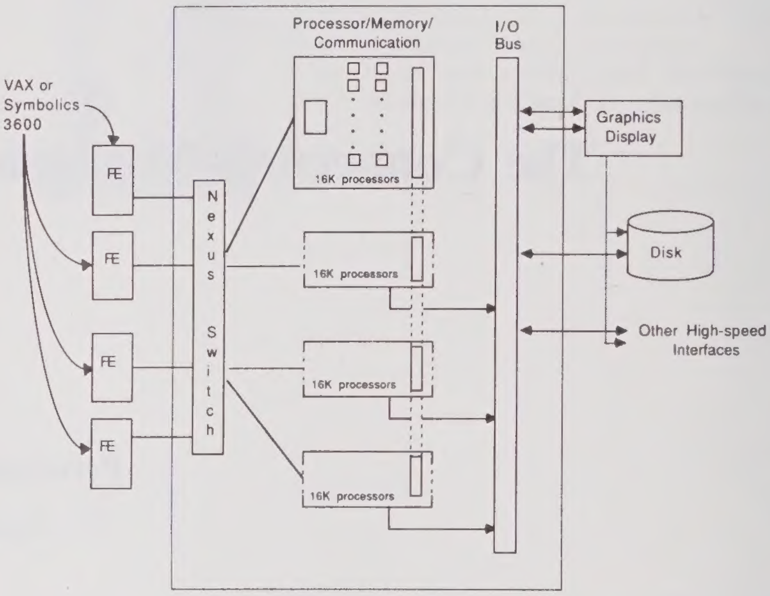


Fig. 1. Connection Machine system.

II. Hardware Overview

The CM-1 system consists of a processor array, from one to four front-end computers, and high-speed peripherals such as disks and image devices (see Fig. 1). The processor array contains 64000 processing elements, each a simple serial processor with 4000 bits of memory giving 32 Mbytes of total memory for the machine. A full CM-1 system is made up of four sections of 16000 processors each. The sections can be used separately, in pairs, or as one 64 kbyte processor unit.

Aside from its processors, each section contains interprocessor communication hardware, a sequencer, and, optionally, a set of peripherals. The interprocessor communication hardware is used for communication between any processors controlled by one host. A sequencer receives macroinstructions from the front end and broadcasts sequences of microinstructions to all the processors in its section. A typical macroinstruction would be to add two 32-bit numbers and store them in a particular location. Because of the simplicity of the processors, each macroinstruction is typically implemented by many microinstructions.

The front end computer, or host, attaches to the microcontroller through a bidirectional crossbar switch called a

---

Manuscript received April 27, 1987; revised October 31, 1987.

The authors are with Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1280.

IEEE Log Number 8427235.

® Connection Machine is a Registered service mark of Thinking Machines Corporation.

0018-9472/89/0700-0707$01.00 ©1989 IEEE

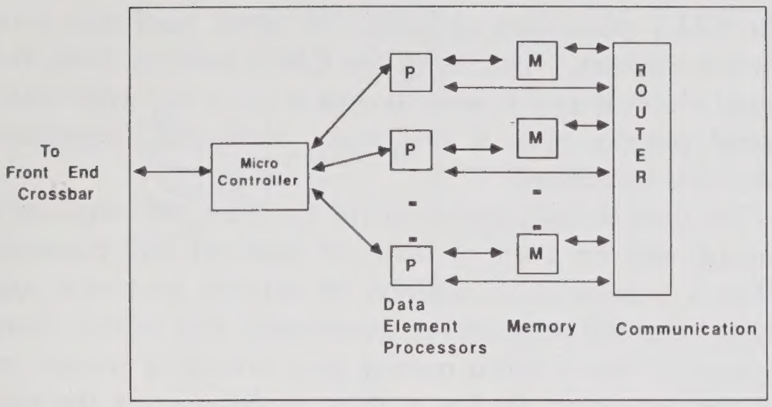


Fig. 2. Processor/memory/communication system.

Nexus, and control one, two, or four 16-kbyte processor sections. Typical front ends are the VAX® or the Symbolics 3600® Series computer.

Although the front end can be used to supply data to the CM-1 computer, in many applications the CM-1 processors can process data much faster than the front end can supply it. For this reason, the CM-1 processors are connected to a high-speed bidirectional bus consisting of up to eight I/O channels, each with a bandwidth of 320 Mbits/s. Special disk drives, frame buffers, frame grabbers, and specialized I/O devices can effectively use this bandwidth. This large I/O bandwidth makes the machine effective in very large data-intensive problems such as data bases, image analysis, and graphics.

Pointer referencing (or referencing the data in one data object from another) on the CM-1 computer requires interprocessor communication. Since a CM-1 processor needs to be able to access data from any other processor, this intercommunication system has to handle a large load at high speed. This is achieved by the use of a router which is integrated into the architecture so that every processor's memory is easily accessible to every other processor. The result is that the application programmer does not have to worry about physical processor geometry since the router handles all interprocessor communication efficiently regardless of layout (see Fig. 2).

Along with general pointer referencing, two-dimensional interprocessor communication is supported. In this type of communication, each data object can communicate with its two-dimensional neighbor (to the north, east, west, or south). This intercommunication is handled by a slightly different mechanism and is faster than using the general communication mechanism. Image processing applications often uses this ability to move data from one pixel to the next. The performance of these individual elements will be treated later in this paper.

III. Software Model

From a programming point of view, the CM-1 system can be viewed as a smart memory that can be addressed and controlled by a front-end computer. The control of the program lies solely in the front end, while data are distributed between the front end and the CM-1 computer, depending on where it can be operated on most efficiently.

This section is intended as an introduction to programming the CM-1 computer. It will describe how data map onto the CM-1 processors and the types of operations that can be performed on CM-1 data. Finally, this section will present an example application and show how it can be implemented on the CM-1 system.

A. The CM-1 Computer as Smart Memory

At the heart of any large problem is the data set consisting of some combination of interconnected data objects such as numbers, characters, records, structures, and arrays. In any application these data must be selected, combined, and operated on. Data parallelism takes advantage of the parallelism inherent in large data sets.

To make effective use of data parallelism, the CM-1 computer has to be made to "look" like the data of the problem being solved. To do this, the processors in the machine are each assigned one data object, which can be any combination of immediate data and pointers. Operations can be specified to operate simultaneously on any or all data objects in the machine.

The CM-1 processors should be used whenever an operation can be performed simultaneously on many data objects. Data objects are left in the CM-1 computer during execution of the program and are operated on in parallel at the command of the front end. This differs from the serial model of reading each data object from its memory one at a time, operating on it, and then storing it again.

The front-end computer handles the flow of control, storage, and execution of the program, and all interaction with the user and/or programmer. The data set, for the most part, is stored on the CM-1 computer. The front end can fetch and store memory in individual processors efficiently to perform serial operations if this is desirable.

There are several direct benefits to maintaining program control only on the front end. First, programmers can work in an environment familiar to them. Programming languages, programming concepts, and program development support (compilers, debuggers, editors, etc.) do not change. Second, most of the existing code for an application does not have to be changed to use the CM-1 processors. A large part of most application programs pertains to the interface between the program, the user, and the operating system. Since the control of the program remains on the front end, code developed for these purposes is useful with or without the CM-1 computer, and only the code pertaining specifically to the data residing on the Connection Machine computer needs to be coded to use the CM-1 processors. Parts of the program which are especially suited for the front end, such as file manipulation, user interface, and serial data operations, can be done on the front end, while the parts of the program that run efficiently on the CM-1, namely the "inner loop" that operates on the data set, can be done there. In this way, the individual strengths of both the serial front end and Connection Machine computer can be maximized.

® VAX is a Registered service mark of Digital Equipment Corporation.
® Symbolics 3600 is a Registered service mark of Symbolics, Inc.

Most large problems have a large data set composed of interconnected structures, arrays, and records which can be mapped directly onto CM-1 processors. *Each processor's memory in the CM-*1 *computer holds the data from one record or array element and some small stack space for temporary results*. A processor can hold data of different types such as integers, floating point numbers, symbols, or pointers. Typically, each processor only needs 128–1024 bits for this information. Rarely is running out of memory within one processor a problem because each physical processor contains 4000 bits of memory.

Problems of different sizes will use different numbers of processors. Some problems will allocate and deallocate processors during the course of running the program. Because the CM-1's flexibility this works well. If a program needs fewer than 64000 processors, then some sit idle, ignoring the instructions being sent by the front end. If a program needs more assigned processors than 64000, then many more "virtual processors" are available (see Section VI).

Examples of data objects used in the CM-1 computer and pixels, transistors, data base records, atoms, and sections of physical space. Each of these are allocated one per processor. (Throughout this paper the terms "processor," "data object," and "record," are used interchangeably.)

B. Program Example

Consider the problem of manipulating a data base of employee records where each employee is a structure of employee ID, title, salary, a pointer to their boss, and a pointer to their assistant (if they have one). The example operations are "give each manager that earns over $100000 a pay cut of $5000," give all managers' assistants a ten percent raise," and "sum all the managers' salaries" to illustrate selection and parallel operation, pointer reference, and global operations respectively.

For this sample problem, each employee's record would be assigned to a processor. On a serial machine, data are loaded from a file and put into records in the virtual memory of the serial machine. On the CM-1 system, the difference is the records are kept on the CM-1 computer.

The language *Lisp® pronounced star-lisp) is used in this paper to describe the pieces of code for the CM-1 computer. All examples can run on the CM-1 computer. In *Lisp the employee structure might look like this:

```
(*defstruct (employee))
   employee-ID;  type: number
   position;     type: number
   salary;       type: number
   boss;         type: pointer
   assistant;    type: pointer.
```

This declaration indicates that structures of this type should be created and manipulated in the CM-1 processors. If the creation and manipulation were done with the serial front end, then these operations would take about as long using the CM-1 processors as using the serial machine's own virtual memory. Thus using the CM-1 memory from the serial machine as if it were its own is not a very significant speed penalty even if one never used the Connection Machine processors.

The data in each processor (in this case, an employee's record) will be a set of fields of numbers and pointers. That a pointer is an address of another processor just means that the program can get or send data to that other processor; this is called moving data through a pointer, or pointer reference. So far, nothing is different in the way the CM-1 computer feels or performs from programming the front end. In fact most programs look very much the same whether the data are on the CM-1 computer or not (see Fig. 3).

Consider the first operation, to select all managers that earn over $100000 and decrease their salary $5000. To execute this we must select all the data objects that satisfy the predicate "is a manager and makes over $100000." This selection is done by instructing each processor to be "active" when the value in the employee-position slot is equal to *manager* and the value in the employee-salary slot is greater than $100000. Notice that *manager* is a value known by the front-end and is broadcast to each processor to compare against its local data. In *Lisp the front-end code might look like this:

```
(*when (and ( = !! position (!! *manager*))
            ( > !! employee-salary 100000)))
   (*set employee-salary (-!! employee-salary (!! 5000))).
```

Executing this code on the front end would deliver an instruction stream that would look something like:

```
(let ((saved-context-bit *stack-index*))
   (CM:store-context-always saved-context-bit)
   (CM:u= constant (pvar-location position) *manager*
    8)
   (CM:and-context-with-test)
   (CM:u> constant (pvar-location employee-salary)
    100000 20)
   (CM:and-context-with-test)
   (CM:u-constant (pvar-location employee-salary) 5000
    20)
   (CM:load-context-always saved-context-bit)).
```

Notice that these instructions look like a serial computer's macroinstruction set. We have selected a set of processors and executed an operation within those processors.

To perform the "give each managers' assistant a ten-percent raise" instruction requires communication between processors. This is done by referencing data in the processor pointed to by a field in another processor. In other words, a processor would have an address (pointer) in one of its slots to another processor.

This operation is done in the following steps:

- each manager's processor gets the salary information from their assistant (in the program this is "pref!!"; or parallel reference);

®*Lisp is a Trademark of Thinking Machines Corporation.

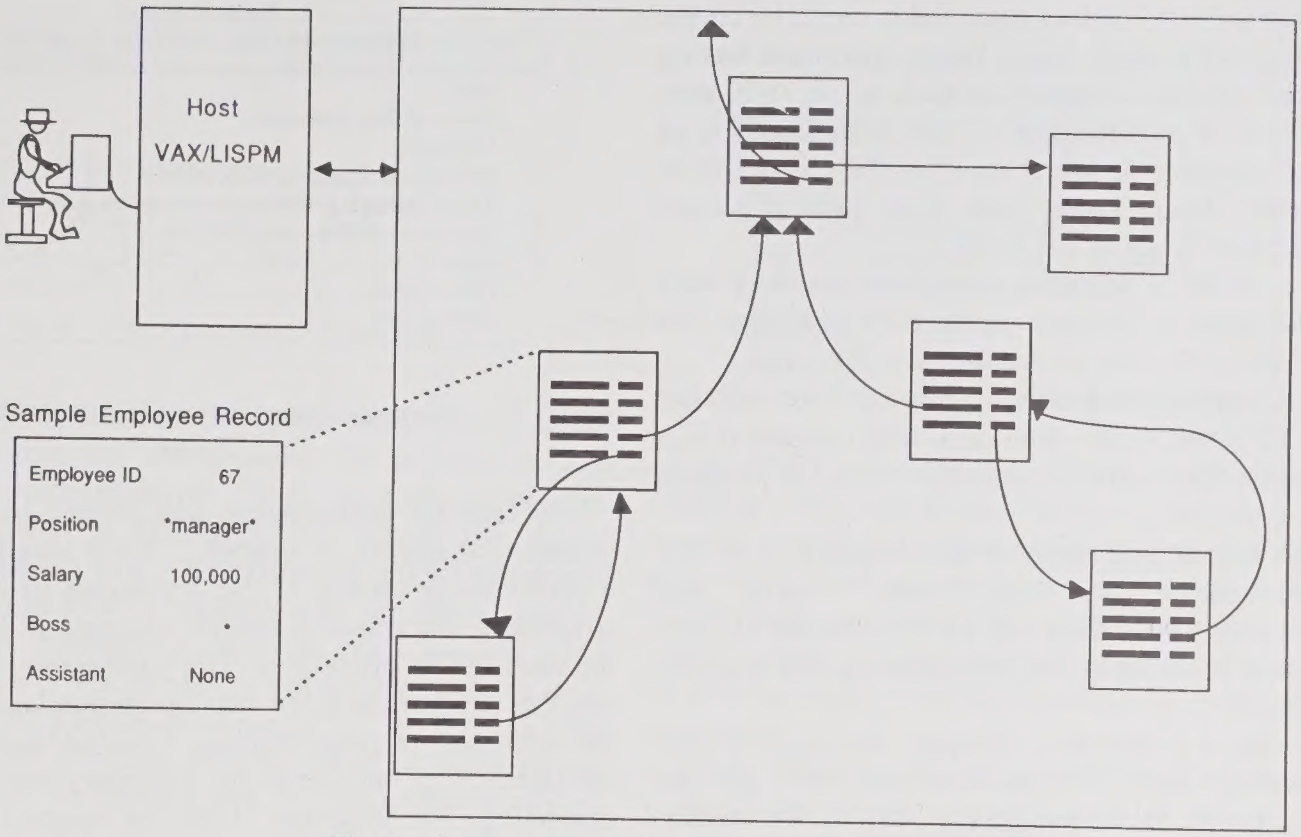


Fig. 3. Connection Machine processors: software view.

- the manager's processor saves that information in a temporary location (this is a move with the processor to the stack (the "*let"));
- the new salary is computed by multiplying the old one by 1.10 in each manager's slot (the "*!!");
- the result is sent back to the assistant's processor and put in the salary slot. This involves another move through a pointer. (This is the pset, or memory parallel set.)

This operation is done, in *Lisp, as

```
(*let (assistants-salary!! (pref!! assistant employee-
   salary))
   (*pset assistant employee-salary
      (*!! assistant-salary!! (!!0.10)))).
```

The operation, "total all managers' salary," returns a number to the front-end machine which is the total of all managers' salary slots. This is done by selecting all managers and then doing a *global sum*. Global sum is implemented using a logarithmic time tree reduction using the intercommunication wires directly. In *Lisp this looks like:

```
(*when ( = !! position (!! *manager*))   ;select managers
(*sum employee-salary))                  ;return sum.
```

This is an example of the use of *scan* that is discussed in the next section. This example shows how a specific example might map onto the Connection Machine processors and how common operations are used on such data.

IV. Basic Performance of the CM-1 System

The performance of the CM-1 system depends on the relative speeds of different common operations. Each application uses a different mix of instructions, so exact application performance depends on processor utilization and the weighted average of the instructions used. The performance rates shown in Table I describe 32-bit operations from the point of view of an active processor as a ratio to the speed of an add. Further, this section describes how these numbers scale with different arithmetic precisions and "virtual processor" ratios.

TABLE I
Relative Times of Various 32-Bit Operations

Add	1
Move within processor	0.6
Multiply	32
Move with nearest grid neighbor	4
Move through pointer (pointer reference)	25
I/O move (our or into the CM-1)	20
Scan	25
Global mix	2
Global sum	25

Performance per processor is the operation speed of an active processor or data object. A 32-bit integer addition, for instance, takes about 32 μs per processor. The system performance depends on both the speed per processor and the number of active processors. Thus the peak system performance in additions is about 2000 million instructions per second (MIPS). Other elements that influence performance are the instruction mix used by the application, the precision of the arithmetic and the total number of processors used. Some applications, such as image processing and cellular automata, run significantly faster than 2000 MIPS because they commonly use instructions on less than 32 bits. Table I uses 32 bits as a unit of measurement for other performances.

Table I describes the machine as a set of ratios with add time. Each entry is described in the following.

The 32-bit add is the unit in Table I. An add is a typical logical operation where the operands and results stay within each processor. Each of these fields might be on the stack or a slot in the data object. Other operations having similar speeds are logical operations such as OR, AND, XOR, etc. Also, if one of the operands to these instructions is an "immediate" number (a literal number that comes from the instruction stream rather than from local processor memory) the time is about the same.

The move within a processor command moves a field from one location to another within each processor. An example of this is moving an argument to the stack.

Multiply is implemented with a "shift-add" for each bit, so it takes 32 times longer than each add because it is a 32-bit multiply. This corresponds to the *!! in the program example.

Move with nearest grid neighbor are transfers of 32 bits with the processor to the "east," "west," "north," and "south." All active processors will do this transfer at once. This operation is useful in image processing and graphics applications.

Move through a pointer is a reference into another processor's memory. This corresponds to the pref!! (get the assistants' salaries) in the program example. The performance of this operation varies only slightly with the layout of the processors. Of course, if the problem is a grid problem, then the grid communications should be used instead.

I/O move is the time required to get the I/O port on the CM-1 computer. The real I/O performance will depend on the I/O device being used because most will not sustain a 320-Mbit/s/8000 processor rate. The important aspect in this performance is its balance with the other performances so that the CM-1 computer can be connected directly into data-intensive applications.

Scan is an operation that performs a form of processor intercommunication very quickly. Add–scan, for instance, results in each processor getting the sum in all processors before it. This is implemented with a parallel algorithm. Max–scan leaves the maximum up to that point:

Processor number	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	0	0	1	1	2	1	0	1
Result of add-scan	0	1	2	2	2	3	4	6	7	7	8
Result of max-scan	0	1	1	1	1	1	1	2	2	2	2.

This is a useful operation on the CM-1 computer for implementing other instructions such as global-sum but many uses have little analogy with serial software cliches.

Global-max results in the maximum value held by any active processor being passed back to the front end computer. The speed of this instruction is due to special hardware. Executing an AND of a field in all processors uses the same mechanism and runs at approximately the same speed. Similarly, OR and MINIMUM perform about the same.

Global-sum is the sum of all values held by processors in the CM-1 computer. This is implemented using *add – scan* and taking the highest processor's result.

TABLE II
RELATIVE TIMES FOR OPERATIONS ON N-BIT NUMBERS

Add	N
Move within processor	N
Multiply	N^2
Move with nearest grid neighbor	N
Move through pointer (pointer reference)	N
I/O move (out or into the CM-1)	N
Scan	N
Global max	N
Global sum	N

V. PERFORMANCES WITH DIFFERENT WORD LENGTHS

The effective performance will change based on what precision arithmetic is needed. "Word length" is used to describe the precision of the arguments to the CM-1 instructions. The previous section assumed 32-bit arithmetic for each of the operations. The Connection Machine system can operate on a few bits up to hundred of bits with the same ease or programming. Variable length arguments are handled in microcode for efficiency and ease of programming. The language *Lisp, for example, can extend the word length of numbers as needed. Sometimes a program will need only 16-bit numbers to start, but will use 32 or 64-bit numbers later (or 53 bits for that matter). It is, of course, desirable to minimize the precision required to maximize performance and minimize space utilization.

As one can see in Table II, most operations scale linearly with the word length. Thus if only 16 bit additions are needed, twice as many can be done in the time of a 32-bit addition. There is a small amount of calling overhead in each instruction call, so behavior is not quite linear for very short numbers.

Multiplication uses a shift–add algorithm so that the precision scales with the square of the number of bits; thus a 16-bit multiply would take 1/4 the time of a 32-bit multiply. The CM-1 system's flexibility extends to the precision of the arithmetic needed. Image processing applications often use very limited precision numbers, and numerical simulation often use very large precision numbers.

VI. VIRTUAL PROCESSOR CONCEPT

As presented so far, the CM-1 system has a constant (independent of problem size) performance of an add operation for problem sizes up to 64000 processors. In real applications, often 64000 processors are not enough. For example, an image processing application with a 1000× 1000 image would require one million processors, one for each pixel. To get around the arbitrary limit of the number of physical processors, the CM-1 supports virtual processors (see Fig. 4). Virtual processors are a software abstraction, implemented at the microcode level, which allows a programmer to write programs that are independent of the number of physical processors that the CM-1 hardware contains. This section will describe how virtual processors are implemented on the CM-1, and what effect they have on program performance.

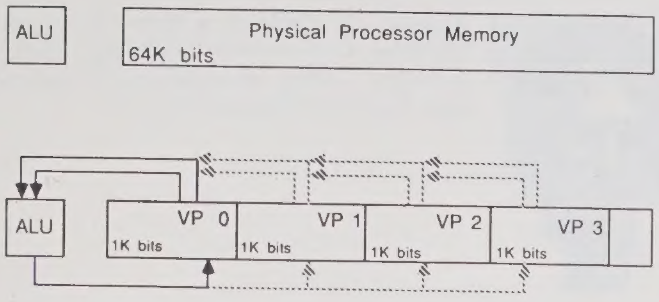


Fig. 4. Virtual processor (VP) concept.

Virtual processors are implemented using three separate mechanisms: one for storage, one for processing, and another for communications. First, the memory of each physical processor is divided evenly among the virtual processors assigned to it. The number of virtual processors per physical processor is referred to as the virtual processor ratio. For example, a problem requiring 1M (2^{20}) virtual processors would need a virtual processor ratio of 16 to operate on a 64000-machine ($2^{20}/2^{16} = 2^4 = 16$). This means that the physical address space of each virtual processor would be divided into 16 equal address spaces, one for each virtual processor assigned to it. Image processing applications commonly use only 100 bit/virtual processor so 32 processors can be assigned to each physical processor, and the CM-1 computer can be configured as a two million-processor machine.

The second mechanism necessary to support virtual processors is time-multiplexing of the physical processors among the virtual processors assigned to it. Every macroinstruction sent by the front end is run on each of the virtual processors within each physical processor. The overhead for switching context is extremely small (about the time to execute a 2-bit add) because each processor is so simple. In the 1M processor example given above, each macro instruction would be executed 16 times by each physical processor.

The third mechanism of communications allow the CM-1 processors to communicate with one another without regard to virtual processors. Grid communications are handled by the microcode by sharing the grid wires. General communications (pointer reference) is handled by the router hardware. The length of a processor address changes based on the number of virtual processors in the machine. This virtual address is used by the router hardware to deliver messages to the correct virtual processor.

Since the virtual processor abstraction is implemented at the microcode level, the virtual processing mechanism is both efficient and totally transparent to the programmer. These characteristics make the CM-1 system appropriate for problems of very different sizes. In general, it is not necessary to know how many physical processors are in a CM-1 computer.

A. Effect of Virtual Processors on Performance

Since a physical processor is shared among several virtual processors, the performance from a virtual processor's point of view goes down. If one measures the performance of the CM-1 computer as a whole, however, in terms of MIPS or MFLOPS, the performance does not go down but remains constant. In other words, given a virtual processor ratio of 16, each virtual processor is only getting 1/16 of the physical processors time, so that each macroinstruction takes 16 times as long, but each macroinstruction is doing 16 times as much work as in the case of a virtual processor ratio of 1.

TABLE III
RELATIVE TIME OF OPERATIONS WITH K VIRTUAL PROCESSORS ($K = 1000$)

Add	K
Move within processor	K
Multiply	K
Move with nearest grid neighbor	K
Move through pointer (pointer reference)	$K \log K$
I/O move (out or into the CM-1)	K
Scan	K
Global max	K
Global sum	K

Table III shows the effects of virtual processors on the performance of the various classes of macroinstructions. As expected, most operations slow down linearly with the virtual processor ratio. The only exceptions are those that involve interprocessor communication which slow by $O(K \log(K))$ for virtual processor ratios less than 16. The performance for very high ratios has not been studied. The addition of the $\log(K)$ term is due to the increase in router congestion.

Virtual processors free the programmer from the physical size of the CM-1, and also from the size of the data set that the program is to operate on. Since this is done at a very low cost in speed and programming, virtual processors have been used in most applications programmed for the CM-1 system.

VII. CONCLUSION

The model CM-1 system offers a high level of performance due to its flexible massively parallel architecture. Much of the maximum peak performance of 2000 MIPS can be used on a problem due to the balanced performances of all common instructions. The CM-1 can be used for very different types of applications because the machine can be efficiently reconfigured to have the needed number of processors and the needed arithmetic precision.

Since the CM-1 computer works closely with a serial front end computer, the programmer operates in a familiar environment. The CM-1 system combines the benefits of programming the front end computer with the performance of a massively parallel machine.

REFERENCES

[1] Bolt Beranek and Newman, Inc., "Development of a butterfly multiprocessor test bed," Rep. 5872, Quarterly Tech. Rep. 1, Mar. 1985.

[2] A. Gottlieb *et al.*, "The NYU ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, pp. 175–189, Feb. 1983.

[3] W. D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[4] "Introduction to Data Level Parallelism," Thinking Machines Tech. Rep. 86.14, Apr. 1986.

[5] C. Lasser and S. Omohundro, "The essential *Lisp manual," Thinking Machines Corp., July 1986.

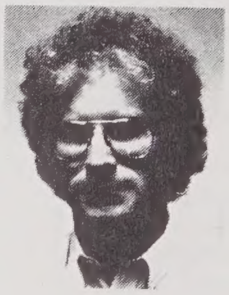
[6] D. H. Schaefer, J. R. Fischer, and K. R. Wallgren "The massively parallel processor," *Eng. Notes*, vol. 5, no. 3, pp. 313–315, May–June 1982.

[7] H. J. Siegel *et al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, Dec. 1981.

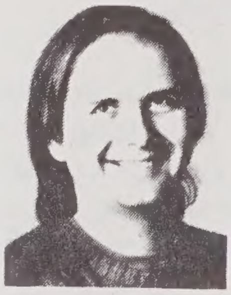
[8] D. E. Shaw, "The NON-VON Supercomputer," Dep. Comput. Sci., Columbia Univ., New York, NY, Aug. 1982.

[9] S. J. Stolfo and D. E. Shaw, "DADO: A tree-structured machine architecture for production systems," Dep. Comput. Sci., Columbia Univ., New York, NY, Mar. 1982.

[10] P. M. Flanders *et al.*, "Efficient high speed computing with the distributed array processor," in *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. New York: Academic, 1977, pp. 113–127.



Brewster A. Kahle is a Scientist at Thinking Machines Corporation, Cambridge, MA. He is currently working on architectures and applications for the Connection Machine.



W. Daniel Hillis is a Founding Scientist of Thinking Machines Corporation, Cambridge, MA, currently working on machine learning and parallel computing. He is the principal architect of the Connection Machine computer.

Mr. Hillis won a 1985 ACM Distinguished Thesis Award for the book *The Connection Machine*.

Speech Partial Papers

Smead ®
No. 153L
HASTINGS, MN